Universität Bremen
FB 3 – Informatik
Prof. Dr. Rainer Koschke
Tutor: Dierk Lüdemann

Software–Projekt 2 2012/13

VAK 03-BA-901.02

Architekturbeschreibung

Abgabe: 23. Dezember 2012 — Version 1.0

Inhaltsverzeichnis

Abbildungsverzeichnis

Version und Änderungsgeschichte

Version	Datum	Änderungen
0.1	20.11.2012	Dokumentvorlage als initiale Fassung kopiert
1.0	23.12.2012	Architekturbeschreibung in erster Version fertig

1 Einführung

1.1 Zweck (Rhea)

Dieses Dokument dient der Beschreibung der Architektur. Diese ist wichtig für die Entwicklung des Systems, die Entwickler entnehmen ihr die Funktionalität der einzelnen Komponenten. Es ist wichtig eine präzise Planung der Architektur zu erstellen, um die Schwierigkeiten und die Möglichkeiten die wir haben, herauszufiltern und zu beeinflussen. Dafür müssen das Zusammenspiel der verschiedenen Komponenten detailliert herausgearbeitet und beschrieben werden, Einflussfaktoren und Abhängigkeiten herausgearbeitet werden und Probleme mit Lösungsstrategien entwickelt werden.

Die Architektur ist das Grundgerüst für unsere Implementierung, und daher hängt diese auch direkt von der Architektur ab. Daraus folgt auch, je genauer und besser die Architektur, desto einfacherer und besser wird auch die Umsetzung der Implementierung.

1.2 Status

Diese Beschreibung in der aktuellen Version (1.0 vom 23.12.2012) beschreibt den ersten Entwurf der Architektur.

1.3 Definitionen, Akronyme und Abkürzungen (Thomas & Lukas)

1.4 Referenzen (Thomas & Lukas)

Rainer Koschke: SWP2: Vorlage zu diesem Dokument. (*Abruf 23.10.2012*)

Rainer Koschke: Mindestanforderungen SWP 2. http://www.informatik.uni-bremen.de/st/Lehre/swp/mindestanforderungen.html (*Abruf 10.11.2012*)

Rainer Koschke: SWP1: Vorlesung und Folien zu Modellierung (*Abruf 17.12.2012*)

Rainer Koschke: SWP1: Vorlesung und Folien zu Softwarearchitektur (*Abruf 10.12.2012*)

Wikipedia: Lernkartei. http://de.wikipedia.org/wiki/Lernkartei (*Abruf am 15.11.2012*)

Wikipedia Eintrag zu HTTP http://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol (*Abruf 10.12.2012*)

Wikipedia Eintrag zu HTTPS http://de.wikipedia.org/wiki/Https (*Abruf 10.12.2012*)

Wikipedia Eintrag zu SSL http://de.wikipedia.org/wiki/Transport_Layer_Security (*Abruf 10.12.2012*)

Die Java Programmiersprache http://java.com (*Abruf 17.11.2012*)

Wikipedia Eintrag zu JavaEE http://de.wikipedia.org/wiki/Java_Platform,_Enterprise_Edition (*Abruf 12.12.2012*)

Wikipedia Eintrag zu Servlets http://de.wikipedia.org/wiki/Servlet (*Abruf 12.12.2012*)

Wikipedia Eintrag zu SQL http://de.wikipedia.org/wiki/SQL (*Abruf 07.12.2012*)

The Apache Software Foundation: Apache Derby Project. http://db.apache.org/derby (*Abruf 08.11.2012*)

Projektseite zu MySQL http://www.mysql.com/ (*Abruf 07.12.2012*)

Projektseite zu HyperSQL http://hsqldb.org/ (*Abruf 07.12.2012*)

Projektseite zu SQLDroid https://github.com/SQLDroid/SQLDroid (*Abruf 18.12.2012*)

RFC zu CSV http://tools.ietf.org/html/rfc4180 (*Abruf 17.11.2012*)

Der Jetty Servlet Application Server http://www.eclipse.org/jetty/ (*Abruf 02.12.2012*)

Projektseite zu Tomcat http://tomcat.apache.org/ (*Abruf 02.12.2012*)

Spezifikation zu XMLRPC http://xmlrpc.scripting.com/spec.html (*Abruf 02.12.2012*)

Projektseite zu ActionbarSherlock http://actionbarsherlock.com/ (*Abruf 15.12.2012*)

Projektseite zu Eclipse http://www.eclipse.org/ (*Abruf 23.12.2012*)

1.5 Übersicht über das Dokument (Rhea)

Dieses Dokument basiert auf der Vorlage des IEEE P1471 2002 Standards und behandelt folgende Inhalte:

1. **Einführung**
   In der Einführung wird erklärt wozu und wem dieses Dokument dient. Es werden Worterklärungen und die genutzten Referenzen aufgelistet und ein Überblick über die Inhalte des Dokumentes geliefert.

2. **Globale Analyse**
   Hier werden die Faktoren, die unsere Architektur beeinflussen gesammelt und niedergeschrieben. Es wird herausgearbeitet, was für Auswirkungen diese auf die Architektur haben und aus diesen Faktoren werden dann Probleme abgeleitet. Für jedes Problem werden Lösungsstrategien entwickelt, und die sinnigste Strategie ausgewählt, um das Problem zu lösen.
3. **Konzeptionelle Sicht**
   In der konzeptionellen Sicht werden die Komponenten im groben vorgestellt. Außerdem wird das Zusammenspiel der einzelnen Komponenten dargestellt. Diese Sicht beschreibt auf einer abstrahierten Ebene und beschäftigt sich dadurch weniger mit Details.
4. **Modulsicht**
   Diese Sicht ist eine Ebene tiefer abstrahiert, als die konzeptionelle Sicht. Die Komponenten der konzeptionellen Sicht werden hier in Pakete eingeteilt und diese Pakete werden dann zu neuen Modulen aufgeteilt.
5. **Datensicht**
   Die Datensicht zeigt hier die Abbilder der Datenbank des Clients und des Servers. Zudem wird beschrieben, wie die einzelnen Tabellen und Einträge miteinander Zusammenspielen und einander zugeordnet werden können.
6. **Ausführungssicht**
   In der Ausführungssicht wird beschrieben welche Module bei der Ausführung der Anwendung arbeiten und wie diese zusammenspielen. Sie besteht aus einem Modell und einem erklärendem Text.
7. **Zusammenhänge zwischen Anwendungsfällen und Architektur**
   Die Zusammenhänge zwischen den Anwendungsfällen, die wir in der Anforderungsspezifikation herausgearbeitet haben, und den Designentscheidungen der Architektur, werden hier aufgedeckt.
8. **Evolution**
   In der Evolution wird beschrieben, was wir bei Veränderungen der Rahmenbedingungen ändern würden bzw. ändern müssten.

2 Globale Analyse

2.1 Einflussfaktoren (Jöran & Rhea)

Nummer	Faktoren
0	Mehrsprachigkeit der Applikation
1	Tablet und Smartphone Kompatibilität
2	Einhalten des Corporate Design

3	Speichern und Lesen von Daten
4	Verschlüsselung der Kommunikation
5	Implementierung von polyhierarchischen Bäumen
6	Zusammenführen von Bäumen
7	Adminzugang wird benötigt
8	Zugriff vieler Nutzer zur selben Zeit akzeptieren
9	Import & Export des Datenbestandes
10	Lauffähigkeit des Servers auf allen gängigen Betriebssystemen
11	Fremdbibliotheken dürfen für den Einsatz in Forschung und Lehre keine Beschränkungen haben
12	Testbarkeit der Implementierung
13	Performanz der Implementierung
14	Zuverlässigkeit der Implementierung

Nun folgt eine detaillierte Beschreibung der Faktoren. Sie beinhaltet die Ausarbeitung der Auswirkungen, Flexibilität und Veränderlichkeit.

0	**Mehrsprachigkeit der Applikation**
Auswirkungen • Das Produkt muss mindestens vorbereitet sein, mehrere Nutzersprachen zu unterstützen. • Es sollte schnell und einfach möglich sein, eine neue Sprache hinzuzufügen.	
Flexibilität • Wir sind bei diesem Faktor insofern flexibel, dass wir nur mindestens zwei verschiedene Sprachen unterstützen müssen. Außerdem müssen wir nur die Möglichkeit aufweisen, und diese nicht explizit implementieren.	
Veränderlichkeit • fest • Dieser Faktor ist auf Grund der Mindestanforderungen nicht veränderlich.	

1	**Tablet und Smartphone Kompatibilität**
Auswirkungen	

- Es muss sichergestellt werden, dass die Applikation auf beiden Endgerät-Typen anwendbar ist.
- Falls Funktionen nur auf einem Gerätetypen anwendbar sind, müssen sie für den anderen angepasst werden.

Flexibilität

- Es gibt hier für uns keine Flexibilität, da das Produkt auf Tablets und Smartphones lauffähig sein muss.

Veränderlichkeit

- fest
- Die Veränderlichkeit dieses Faktors ist fest, da es eine Mindestanforderung ist.

2	Einhalten des Corporate Design

Auswirkungen

- Das Corporate Design soll so implementiert werden, wie der Kunde es sich vorgestellt hat.
- Das Design der Applikation ist abhängig vom Corporate Design.
- Die Designmöglichkeiten werden eingeschränkt und die Funktionen der Applikation müssen ggf. an das Design angepasst werden.
- Für die genaue Umsetzung der Designentscheidungen, wie der Kunde sie wünscht, fällt Zeit an, die eventuell in der Implementierungsphase fehlen könnte.

Flexibilität

- Es sollte so früh wie möglich entschieden werden, ob das Design implementiert wird oder nicht (gegeben aus der Veränderlichkeit).
- Eventuell kann durch frühzeitiges Ausarbeiten der Desginvorstellungen des Kunden späterer Zeitmangel und Schwierigkeiten reduziert werden.

Veränderlichkeit

- variabel
- Dieser Faktor ist veränderlich. Er wurde nicht in den Mindestanforderungen gefordert, und kann bei Zeitmangel evtl. weggelassen werden.

3	Speichern und Lesen von Daten

Auswirkungen

- Client sowie Server müssen Daten lokal speichern und lesen.
- Außerdem müssen der Client und der Server untereinander über das Internet Daten austauschen können.

Flexibilität

- Dieser Faktor ist nicht flexibel, da das Produkt, was wir entwickeln Daten speichern und lesen können muss.

Veränderlichkeit

- fest
- Dieser Faktor lässt sich auch während des Entwicklungsprozesses nicht verändern.

4	Verschlüsselung der Kommunikation

Auswirkungen

- Es muss eine Möglichkeit gefunden werden, um die Kommunikation effektiv und sicher zu verschlüsseln, die dann auch wieder performant entschlüsselt werden kann.
- Daher muss nach einer geeigneten Verschlüsselungsmethode gesucht werden. Diese sollte auf die Anwendung abgestimmt sein.

Flexibilität

- Wir sind hier in so fern flexibel, dass die Wahl der Verschlüsselung uns überlassen ist.

Veränderlichkeit

- fest
- Die Veränderlichkeit dieses Faktors ist fest, da es eine Mindestanforderung ist.

5	Implementierung von polyhierarchischen Bäumen

Auswirkungen

- Es muss eine Möglichkeit gefunden werden, die polyhierarchischen Bäume sinnvoll zu implementieren.
- Es müssen sich Gedanken darüber gemacht werden, wie die Polyhierarchie implementiert werden kann.
- Gegebenenfalls müssen die Attribute der anderen Klassen so angepasst werden, dass sie als Baum repräsentiert werden können. (Karten und Kategorien)

Flexibilität

- Dieser Faktor ist nicht flexibel, da es sich bei dem verwenden eines polyhierarchischen Baums für die Struktur unseres Glossars bzw. unserer Lernkartei, um eine Mindestanforderung handelt.

Veränderlichkeit

- fest
- Auf Grund dessen, dass es sich bei diesem Faktor um eine Mindestanforderung handelt, ist er auch nicht veränderlich.

6	Zusammenführen von Bäumen

Auswirkungen

- Wir müssen uns Merging-Strategien für Bäume mit verschiedenen Änderungen bzw. verschiedener Version überlegen.

Flexibilität

- Die Art und Weise, wie unsere Bäume zusammengeführt werden, bleibt uns überlassen.

Veränderlichkeit

- teils variabel
- Bis auf den Ausnahmefall, dass ein bis mehrere Gruppenmitglieder die Gruppe verlassen, gibt es keine Veränderung der Mindestanforderungen und somit auch keine Veränderung des Faktors.

7	**Adminzugang wird benötigt**

Auswirkungen

- Wir müssen eine Lösung finden, wie wir einen Adminzugang und ein Admininterface darstellen und entwickeln.

Flexibilität

- Wir können im Laufe des Projekts noch eine andere Idee realisieren, als wie es ursprünglich geplant war, da es keine Anforderung gibt, die genauer spezifiziert, wie das Admininterface aussehen muss.

Veränderlichkeit

- fest
- Der Faktor ist nicht veränderlich, da es eine Mindestanforderung ist, eine Möglichkeit zu implementieren, wie die Admins auf das laufende Produkt zugreifen können.

8	**Zugriff vieler Nutzer zur gleichen Zeit akzeptieren**

Auswirkungen

- Der Server muss mit vielen gleichzeitigen Anfragen umgehen können.
- Es muss eine Lösung gefunden werden, wie der Server die Anfragen behandelt und ob er sie ggf. nacheinander oder gleichzeitig *beantwortet*.

Flexibilität

- Wieviele Zugriffe zur selben Zeit wir akzeptieren, ist im Grunde genommen uns überlassen, da es keine Vorgaben dafür gibt.

Veränderlichkeit

- variabel
- Die Veränderlichkeit des Faktors ist variabel, da es keine Mindestanforderung ist.

9	**Import & Export des Datenbestandes**

Auswirkungen

- Es muss eine Möglichkeit geschaffen werden den Datenbestand aus der Datenbank zu importieren und zu exportieren.
- Beim Importieren muss darauf geachtet werden, dass die importierten Daten richtig mit den bestehenden Daten zusammengefügt werden.

Flexibilität

- Dieser Faktor ist nicht flexibel. Es ist eine Mindestanforderung den Import bzw. Export von Daten für die Administratoren bzw. den Betreiber zur Verfügung zu stellen.

Veränderlichkeit

- fest
- Die Veränderlichkeit dieses Faktors ist fest, da es eine Mindestanforderung ist.

10	**Lauffähigkeit des Servers auf allen gängigen Betriebssystemen**

Auswirkungen

- Wir müssen einen Webserver-Framework verwenden, welches auf den nötigen Betriebssystemen laufen.
- Bei der Implementierung unseres Admintools, müssen wir darauf achten, plattformunabhängige Mittel zu nutzen, die also auf Linux, MacOS und Windows laufen.

Flexibilität

- Welches Framework wir verwenden können wir noch im Laufe des Entwicklungsprozesses ändern, allerdings, muss der gewählte Server definitiv alle geforderten Betriebssysteme unterstützen.

Veränderlichkeit

- fest
- Ist nicht veränderbar, da es sich bei diesem Faktor um eine Mindestanforderung handelt.

11	**Fremdbibliotheken dürfen für den Einsatz in Forschung und Lehre keine Beschränkungen haben**

Auswirkungen

- Wir können keine Software bzw. Bibliotheken oder Frameworks benutzen, deren Nutzung nicht kostenlos ist.

Flexibilität

- Zwar ist der Wahl der von uns verwendeten Frameworks und Bibliotheken frei, aber sie müssen zwingend für die Verwendung in Forschung und Lehre umsonst zu nutzen sein. Also ist je nachdem welchen Aspekt man betrachtet, dieser Faktor flexibel oder eben nicht.

Veränderlichkeit

- fest
- Hier gilt das Selbe, wie bei der Flexibilität. Die verwendeten Produkte können sich ändern, die Anforderung an sie allerdings nicht.

12	**Performanz der Implementierung**

Auswirkungen

- Das Programm muss einer sinnvollen Architektur unterliegen.
- Der Quellcode muss performant geschrieben sein und unnötiger Quellcode sollte vermieden werden.
- Dieser Faktor wirkt sich auf die Wahl unserer Frameworks aus.

Flexibilität

- Es wurde in den Mindestanforderungen keine Aussage darüber getroffen, wie performant unser Produkt am Ende laufen muss. In der Anforderungsspezifikation haben wir maximale Ausführungszeiten (der Aktionen) selbst festgelegt.

Veränderlichkeit

- teils variabel
- Zwar sollten die in den Anforderungsspezifikationen genannten Werte eingehalten werden, allerdings haben wir die Möglichkeit mit Änderungsdokumenten diese Werte noch zu ändern.

13	**Zuverlässigkeit der Implementierung**

Auswirkungen

- Die Architektur muss so durchdacht werden, dass die spätere Implementierung ein zuverlässiges Programm ergibt.
- Störanfälliger Code muss vermieden werden.
- Verwendung bekannter Technologien und Entwurfsmuster.

Flexibilität

- Da es keine Anforderung gibt, die genauer darauf eingeht, wie zuverlässig unser Produkt sein soll, können wir uns in diesem Belangen eigene Ziele stecken.

Veränderlichkeit

- teils variabel
- Die am Anfang gesteckten Ziele der Zuverlässigkeit der Implementierung sollten eingehalten werden. Da es aber keine vorgegebenen Anforderungen zu diesem Thema gibt, sind Änderungen während des Projektverlaufs möglich.

14	**Möglichkeit neue Lernsystem hinzufügen**
Auswirkungen • Wir müssen in unserer Implementierung Mechanismen einbauen, die es erlauben, immer neue Lernsystem hinzuzufügen.	
Flexibilität • Dieser Faktor ist in seiner Umsetzung flexibel, muss jedoch nach den gegebenen Mindestanforderungen implementiert werden.	
Veränderlichkeit • fest • Dieser Faktor ist nicht veränderlich, da es sich um eine Mindestanforderung handelt.	

Es folgen die **organisatorischen Faktoren**.

Nummer	Faktoren
15	Einhalten der Deadline (Abgabetermin)
16	Einhalten des Budgets
17	Erfahrung des Teams
18	Kenntnisse der Programmiersprache Java
19	Größe des Teams
20	JUnit Tests für unser Produkt
21	Kenntnisse der Android-Programm

15	**Einhalten der Deadline (Abgabetermin)**
Auswirkungen	

- Die Applikation muss bis zum Eintreten der Deadline fertig sein.
- Es müssen Strategien entwickelt werden, die sicherstellen, dass das Produkt vor dem Eintreten der Deadline fertig ist.
- Die Architektur muss vermutlich einfach gehalten werden, damit das Produkt in dem knapp bemessenen Zeitraum fertiggestellt werden kann.

Flexibilität

- Die Deadline ist festgelegt und kann nicht verlegt werden. Dieser Faktor ist daher nur Flexibel in der Zeiteinteilung der Aufgaben.

Veränderlichkeit

- fest
- Es gibt keine Möglichkeit die Deadline zu verschieben.

16	Einhalten des Budgets

Auswirkungen

- Da wir nur ein zeitliches Budget zur Verfügung haben, hat dieser Faktor die Auswirkung, dass wir unsere Zeit gut einteilen müssen.

Flexibilität

- Da uns nur zeitliches Budget zur Verfügung steht, sind wir insofern flexibel, dass wir uns die Zeit selbst einteilen können.

Veränderlichkeit

- teils variabel
- Wir können unser Budget nur in dem Sinne *vergrößern*, dass wir mehr Zeit als angesetzt, für die Fertigstellung des Produktes aufwänden. Weitere Vergrößerung von Ressourcen wird nicht möglich sein.

17	Erfahrung des Teams

Auswirkungen

- Das Team muss Erfahrungen und Fähigkeiten aufweisen, um eine sinnvolle Architektur erstellen zu können.

Flexibilität

- Die Erfahrungen des Teams sind nicht flexibel, da unsere Gruppen fest sind und keine neuen Teammitglieder rekrutiert werden können oder Ähnliches. In der kurz angesetzten Zeit, wird es uns zudem auch nicht möglich sein, durch Schulungen etc. die Erfahrung einzelner Gruppenmitglieder zu erhöhen.

Veränderlichkeit

- fest
- Dieser Faktor ist aus dem selben Grund nicht veränderlich, weshalb er auch nicht flexibel ist.

18	**Kenntnisse der Programmiersprache Java**

Auswirkungen

- Es werden ggf. Schwierigkeiten in der Umsetzung der Implementierung auftreten.
- Fehlende Kenntnisse in Java können dazu führen, dass die Deadline der Implementierungsphase nicht eingehalten werden kann.

Flexibilität

- Dieser Faktor ist nicht flexibel. Unser Produkt muss in Java entwickelt werden. Android-Applikationen lassen sich nur mit Java entwickeln und auch für den Server war es vorgeschrieben, dass alles in Java implementiert wird.

Veränderlichkeit

- fest
- Keine Veränderlichkeit vorhanden, da unser Produkt definitiv eine Android-Applikation sein muss, also zwangsläufig in Java geschrieben werden muss (siehe Mindestanforderungen).

19	Größe des Teams

Auswirkungen

- Das Team muss eine gute Größe haben, um die Aufgaben in all ihrem Umfang zu erledigen.
- Da die Größe eig. nicht veränderlich ist, muss dafür gesorgt werden, dass die Aufgaben sinnvoll aufgeteilt werden.

Flexibilität

- Eigentlich ist dieser Faktor nicht flexibel. Es kann jedoch im Laufe des Projekts passieren, dass Gruppenmitglieder die Gruppe verlassen bzw. die Veranstaltung nicht weiter besuchen. In diesem Fall verringert sich unsere Gruppengröße, wenn keine neuen Mitglieder gefunden werden.

Veränderlichkeit

- teils variabel
- Wir gehen bei diesem Faktor nicht davon aus, dass er sich verändern wird, es kann jedoch nicht ausgeschlossen werden, dass Mitglieder die Gruppe verlassen.

20	JUnit Tests für unser Produkt

Auswirkungen

- Wir müssen während des Architekturentwurfs darauf achten, dass alle Methoden unseres Produkts möglichst einfach zu testen sind.
- Wir müssen am Ende des Projekts vollständiges Testen nachweisen.
- Es müssen verschiedene Arten von Tests durchgeführt werden.

Flexibilität

- Dieser Faktor ist nicht flexibel. Tests müssen definitiv mit abgegeben und erfolgreich durchgeführt werden.

Veränderlichkeit

- fest
- Dieser Faktor ist auf Grund der Mindestanforderungen nicht veränderlich.

21	Kenntnisse der Android Programmierung

Auswirkungen

- Es werden ggf. Schwierigkeiten in der Umsetzung der Implementierung auftreten.
- Auswirkungen auf den Zeitplan in der Implementierungsphase sind wahrscheinlich.
- Es können außerdem Schwierigkeiten auftreten, da Möglichkeiten der Java Programmierung nur zum Teil in der Android Programmierung umsetzbar sind.
- Bei mangelnden Kenntnissen besteht eine eingeschränkte Sicht auf die Möglichkeiten der Android Programmierung und da wirkt sich schon vor der Implementierung auf das Projekt aus.

Flexibilität

- Durch frühzeitiges Auseinandersetzen mit dem Android Betriebssystem können mögliche Implementierungsschwierigkeiten und Möglichkeiten frühzeitig erkannt werden, und bessere Kenntnisse in Java bzw. Android erworben werden.

Veränderlichkeit

- fest
- Da die Veränderlichkeit von 22 fest ist.

Nun werden die **technischen Faktoren** ausgelistet.

Nummer	Faktoren
22	Android als Betriebssystem
23	Kompatibilität mit verschiedenen Android Versionen
24	Vielfalt an Zielhardware
25	Möglichst wenig Datenübertragung per Internet

26	Beschränkte Speichernutzung durch unsere Applikation
27	Nutzung eines SQL Datenbanksystems
28	Möglichkeit des Imports von CSV-Datein

22	**Android als Betriebssystem**

Auswirkungen

- Implementierungsmöglichkeiten sind an die des Android Betriebssystems gebunden und daher eingeschränkt.
- Die Applikation wird nur auf Geräten mit Android lauffähig sein.
- Dieser Faktor wirkt sich auf alle Komponenten der clienteseitigen Applikation aus.

Flexibilität

- Dieser Faktor ist nicht flexibel, da der Kunde ausdrücklich eine Applikation für Android wünscht.

Veränderlichkeit

- fest
- Dieser Faktor ist auf Grund der Mindestanforderungen und der Flexibilität nicht veränderlich.

23	**Kompatibilität mit verschiedenen Android-Versionen**

Auswirkungen

- Bei der Entwicklung müssen die entsprechenden API-Versionen eingehalten werden, um ein Funktionieren auf den verschiedenen Android-Versionen sicherzustellen.
- Es ist schwierig über verschiedene Android-Versionen hinweg ein gleichbleibendes Design zu bewahren.
- Bei Designelemente, die erst für höhere Android-Versionen vorhanden sind, als wir sie unterstützen, beispielsweise die Actionbar, müssen Alternativen gefunden werden.

Flexibilität

- Dieser Faktor ist nur flexibel, wenn sich der Kunde darauf einlässt, unterschiedliches Design der Applikation auf verschiedenen Android-Versionen zu akzeptieren. Sonst gilt, dass die Applikation laut dem Kunden (und der Mindestanforderungen) ab der Android-Version 2.3 lauffähig sein muss.

Veränderlichkeit

- fest
- Dieser Faktor wird sich auf Grund der Mindestanforderungen und der Wünsche des Kundens nicht verändern.

24	**Vielfalt an Zielhardware**

Auswirkungen

- Bei der Implementierung unserer GUI müssen wir die verschiedene Bildschirmgrößen und -auflösungen unterstützen

Flexibilität

- Es ist keine Flexibilität vorhanden, da wir eine Android-Applikation entwickeln und das Android-Betriebssystem auf verschiedener Hardware installiert sein kann.

Veränderlichkeit

- fest
- Keine Veränderlichkeit vorhanden, siehe Flexibilität.

25	**Möglichst wenig Datenübertragung übers Internet**

Auswirkungen

- Die Lerninhalte unserer Applikation, müssen, zumindest zum Teil, offline vorhanden sein.

Flexibilität

- Da es sich bei diesem Faktor um keine Mindestanforderung handelt oder dieser vorgeschrieben war, können wir das Maß hier selber festlegen. Der Faktor ist also flexibel.

Veränderlichkeit

- variabel
- Hier gilt das Selbe, wie für die Flexibilität. Da die Werte bzw. das Maß was *viel* bedeutet und welches Gleichgewicht zwischen Speichernutzung und Datenübertragung richtig ist, von uns bestimmt wird. Daher kann sich dieser Faktor auch über die Zeit verändern.

26	**Beschränkte Speichernutzung durch unsere Applikation**

Auswirkungen

- Es kann nicht zu viel Inhalt der Applikation bzw. unserer Datenbank zu einem bestimmten Zeitpunkt auf dem Gerät gespeichert werden, da sonst zu viel Platz beansprucht wird.

Flexibilität

- Ist insofern flexibel, als dass keine genauen Vorgaben zu der Größe der Applikation gemacht wurden.

Veränderlichkeit

- variabel
- Aus der Flexibilität folgt, dass auch während der laufenden Entwicklungsarbeit Änderungen an der Größe hingenommen werden können. Die Größe der Applikation sollte sich im Rahmen halten, kann sich aber im Grunde genommen noch verändern.

27	**Nutzung eines SQL Datenbanksystems**

Auswirkungen

- Wir müssen uns auf eine bestimmte Datenbank festlegen, die SQL unterstützt.

Flexibilität

- Dieser Faktor ist nicht flexibel, da es eine feste Vorgabe ist eine SQL-Datenbank zu verwenden.

Veränderlichkeit

- fest
- Aus der nicht vorhandenen Flexibilität, folgt auch eine Nichtveränderlichkeit dieses Einflussfaktors.

28	**Möglichkeit des Imports von CSV Datein**

Auswirkungen

- Wir müssen uns überlegen, wie wir die CSV-Datein einlesen und in unsere Datenbank überführen.

Flexibilität

- Dieser Faktor ist insoweit flexibel, dass das Format für den Datenimport und Export nicht vorgeschrieben war.

Veränderlichkeit

- teils variabel
- Der Faktor ist veränderbar, da wir uns im weiteren Verlauf des Projekts theoretisch noch für ein anderes Format entscheiden können.

2.2 Probleme und Strategien (Dominic, Jöran & Rhea)

Nummer	Problem
1	Mangelnde Kenntnisse in der Programmiersprache Java
1.1	Unterschiedliche Java- und Programmierkenntnisse
2	Wenig Erfahrung mit Android-Entwicklung in der Gruppe
3	Einheitliches Design und Verhalten über verschiedene Geräte und Versionen bei zu behalten
4	GUI für Smartphone und Tablet optimieren
5	Hoher Zeitaufwand beim Implementieren der GUI
6	Mangelnde Kenntnisse der Gruppenmitglieder über das Datenbankmanagementsystem

7	In Java geschriebenen Webserver der SSL konfigurierbar ist
8	Verschlüsselte Kommunikation mittels SSL unter Android mit der Android API
9	Zusammenführen der globalen Änderungen und der des einzelnen Nutzers
10	Authentifizierung der Administratoren
11	Änderungen mehrerer Nutzer zur gleichen Zeit behandeln
12	Darstellung der polyhierarchischen Bäume in der Datenbank
13	Zyklenfreiheit der Bäume
14	Import und Export von Daten
15	Möglichkeit der Mehrsprachigkeit garantieren
16	Eingeschränkte Auswahl der Fremdbibliotheken
17	Einhalten der spezifizierten Ladezeiten
18	Behandeln vieler Zugriffe zur gleichen Zeit
19	Einhalten der festgelegten Deadline
20	Probleme in der Gruppenarbeit
21	Blockieren der GUI

1	**Mangelnde Kenntnisse in der Programmiersprache Java**
Die Gruppenmitglieder haben zu wenig Kenntnisse in der Programmiersprache Java. Dadurch, dass unsere Applikation in Java geschrieben wird, stellt dies ein Problem da, denn mit nicht ausreichenden Kenntnissen kann nicht die volle Funktionalität von Java ausgeschöpft werden. Sei dies zum einen durch Unwissen über die verschiedenen Möglichkeiten die Java bietet oder zum anderen über das Nicht-fähig-sein diese umzusetzen.	
Involvierte Einflussfaktoren	

- Produktfaktoren:
  - 4. Verschlüsselung der Kommunikation
  - 5. Implementierung von polyhierarchischen Bäumen
  - 6. Zusammenführen von Bäumen
  - 12. Testbarkeit der Implementierung
  - 13. Performanz der Implementierung
  - 14. Zuverlässigkeit der Implementierung
- Organisatorische Faktoren:
  - 17. Erfahrung des Teams
  - 18. Kenntnisse der Programmiersprache Java
  - 19. Größe des Teams
  - 21. Kenntnisse der Android-Programmierung
- Technische Faktoren
  - 21 Android als Betriebssystem

Strategien

- S1: Es wird eine einfache Architektur entworfen, wo viel mit Frameworks und vorgefertigten Bibliotheken gearbeitet wird, denn jede eigene Implementierung erhöht das Risiko auf Fehlerhaftigkeit eben dieser und das Rad muss nicht neu erfunden werden.

Verwandte Strategien

- Keine

1.1	**Unterschiedliche Java- und Programmierkenntnisse**
Die verschiedenen Gruppenmitglieder haben unterschiedliche Erfahrung mit Java-Programmierung bzw. Programmierung im Allgemeinen.	
Involvierte Einflussfaktoren	

- Produktfaktoren:
  - 4. Verschlüsselung der Kommunikation
  - 5. Implementierung von polyhierarchischen Bäumen
  - 6. Zusammenführen von Bäumen
  - 12. Testbarkeit der Implementierung
  - 13. Performanz der Implementierung
  - 14. Zuverlässigkeit der Implementierung
- Organisatorische Faktoren:
  - 17. Erfahrung des Teams
  -
  - 18. Kenntnisse der Programmiersprache Java
  - 21. Kenntnisse der Android-Programmierung
- Technische Faktoren:
  - 21 Android als Betriebssystem
  - 27. Nutzen eines SQL Datenbanksystems

Strategien

- S1: Die Aufgaben sollten so verteilt sein, dass die Erfahrungen und Kenntnisse der Gruppenmitglieder optimal ausgenutzt werden. Das heißt, dass die Gruppenmitglieder, die Java gut beherrschen, auch den Entwurf der kritischen Teile in der Architektur und die Implementierung übernehmen sollten.

Verwandte Strategien

- S1
  - 2. Strategie S2
  - 6. Strategie S1

2	**Wenig Erfahrung mit Android-Entwicklung in der Gruppe**

Da wir eine Android-Applikation entwickeln, müssen wir die Android-API verwenden können und kennen. Da wir allerdings alle keine Erfahrung mit Android-Entwicklung haben, kennen wir uns wohlmöglich nicht ausreichend aus.

Involvierte Einflussfaktoren

- Produktfaktoren
  - 1. Tablet und Smartphone Kompatibilität
  - 12. Testbarkeit der Implementierung
  - 13. Performanz der Implementierung
  - 14. Zuverlässigkeit der Implementierung
- Organisatorische Faktoren
  - 17. Erfahrung des Teams
  - 21. Kenntnisse der Android-Programmierung
- Technische Faktoren
  - 22. Android als Betriebssystem
  - 23. Kompatibilität mit verschiedenen Android Versionen
  - 24. Vielfalt der Zielhardware

Strategien

- S1: Bevor wir uns in der Architekturbeschreibung auf eine Architektur einigen, erstellen wir Prototypen, welche die Komponenten einzeln und in Komposition testen, um uns damit vertraut zu machen.
- S2: Wir verteilen die Aufgaben anhand der Erfahrungen der Gruppenmitglieder, das heißt, dass wenn sich ein Gruppenmitglied mit einer Sachen bereits auseinandergesetzt hat, dass dieses Mitglied beim Entwurf der Architektur als Ansprechpartner fungiert. Außerdem werden wir eher uns bekannte Technologie verwenden, um Einarbeitungszeit in neue und Fehlentscheidungen zu vermeiden.

Verwandte Strategien

- S1
  - 6. Strategie S2
  - 7. Strategie S1
  - 8. Strategie S1
- S2
  - 1.1. Strategie S1
  - 6. Strategie S1

3	Einheitliches Design und Verhalten

Da Android auf sehr vielen verschiedenen Geräten installiert ist, ist die Zielgruppe unseres Produkts wahrscheinlich stark fragmentiert. Das bedeutet, wir müssen beim Entwerfen und Implementieren unserer Applikation darauf achten, dass unser Design und unsere Grafiken auf verschiedensten Bildschirmauflösungen und Größen gleich bzw. möglichst ähnlich aussehen. Zusätzlich bietet Android Designelemente, die nur ab bestimmten API-Levels unterstützt werden. So kann zum Beispiel die Actionbar erst ab der Android-Version 4.0 verwendet werden können.

Involvierte Einflussfaktoren

- Produktfaktoren:
  - 2. Einhalten des Corporate Design
- Organisatorische Faktoren:
  - 17. Erfahrung des Teams
  - 21. Kenntnisse der Android-Programmierung
- Technische Faktoren:
  - 22. Android als Betriebssystem
  - 23. Kompatibilität mit verschiedenen Android-Versionen
  - 24. Vielfalt an Zielhardware

Strategien

- S1: Wir verwenden in unserer Applikation eine Activity für mehrere Anwendungsfälle. Beispielsweise benutzen wir für die Erstellung einer Kategorie und für die Bearbeitung einer Kategorie die selbe Activity, dies erhöht den Grad der Vereinheitlichung in unserer Applikation.
- S2: Wir entwerfen ein möglichst einfaches Design, dass aus möglichst wenig individuellen Grafiken besteht und außerdem nur GUI-Features verwendet, die auch in der niedrigsten der unterstützten Versionen vorhanden sind.
- S3: Um Designelemente wie die Actionbar auch auf Android-Versionen unter 4.0 zu verwenden, suchen wir Fremdbibliotheken mit denen sich die Actionbar auch ohne direkte Unterstützung durch die Android-API realisieren lässt.
- S4: Wir implementieren die nicht auf allen Versionen unterstützen Features selbst.

Wir haben uns dafür entschieden Strategien S1 und S3 zu benutzen. Wir haben uns eine Bibliothek gesucht, die die Funktionalität der Actionbar auf niedrigere Android-Versionen abbildet [1]. Außerdem hilft die Mehrfachverwendung von Activities dabei, ein zusammenhängendes Design zu gestalten.

4	**GUI für Smartphone und Tablet optimieren**
Das Graphical User Interface muss so optimiert werden, dass es auf Smartphones und Tablets ansehnlich aussieht.	
Involvierte Einflussfaktoren • Produktfaktoren: – 1. Tablet und Smartphone Kompatibilität • Technische Faktoren: – 24. Vielfalt an Zielhardware	
Strategien	

[1] http://actionbarsherlock.com/

- S1: Da eine Android Applikation von vornherein Smartphone und Tablet kompatibel ist, also für generelle Funktion und standardmäßiges Aussehen nicht weiter optimiert werden muss, kann die Optimierung fürs Tablet (aus zeitlichen Gründen) zurückgestellt werden.
- S2: Wir können eine GUI entwerfen, die sowohl auf dem Smartphone als auch auf Tablets ohne Anpassung ansehnlich aussieht.
- S3: Die von Android bereitgestellten Elemente für unsere Applikation nutzen, die dafür optimiert sind, unterschiedliches Design bei unterschiedlicher Bildschirmgröße zu liefern (z.B. Fragments).

Wir entscheiden uns für eine Mischung aus allen 3 Strategien. Wir verwenden wo es geht, die in S3 angesprochenen Elemente. Außerdem ist unsere GUI für eine Lernkartei-Applikation auf dem Tablet, wie auf dem Smartphone "schön anzusehen". Aus zeitlichen Gründen werden aber weitere Optimierungen zurückgestellt.

Verwandte Strategien

- Keine

5	**Hoher Zeitaufwand für die Implementierung der GUI**

Für die Implementierung der GUI wird ein hoher zeitlicher Aufwand anfallen.

Involvierte Einflussfaktoren

- Produktfaktoren
  - 4. Einhalten des Corporate Design

Strategien

- S1: In regem Kontakt mit dem Kunden stehen, um so oft Feedback über den Status des Produktes zu bekommen. Dieses Feedback nutzen um wenig Zeit mit falschen Details zu verschwenden.
- S2: Eclipse bietet die Möglichkeit eine Android GUI in *XML* zu erstellen ohne Quellcode schreiben zu müssen. Dies erspart Zeit.

Wir wählen hier die Strategie S1, denn enger Kontakt mit dem Kunden ist sehr produktiv, denn wir können sofort auf die Wünsche eingehen und bei schlechter Umsetzung unsererseits reagieren.
S2 halten wir für eine schlechte Strategie, denn der Quellcode der beim erstellen der GUI von Eclipse raus kommt, ist sehr zu bemängeln. Ob die GUI dann auf Tablet und den verschiedenen Smartphone-Varianten noch ansehnlich aussieht ist fraglich. Hier möchten wir lieber selber die Kontrolle haben, auch wenn es mit mehr Zeitaufwand verbunden ist.

Verwandte Strategien

- Keine

6	**Mangelnde Kenntnisse der Gruppenmitglieder über das Datenbankmanagementsystem**

Nicht alle unserer Gruppenmitglieder haben Erfahrung mit Datenbanksystemen.

Involvierte Einflussfaktoren

- Produktfaktoren
  - 5. Nutzen von HyperSQL als Datenbanksystem
- Technische Faktoren
  - 27. Nutzung eines SQL Datenbanksystems

Strategien

- S1: Die Gruppenmitglieder, die bereits Erfahrungen mit Datenbankmanagementsystemen haben, sollten im Architekturentwurf und der Implementierung die Arbeit daran übernehmen.
- S2: Prototypen mit den verschiedenen zur Auswahl stehenden Datenbanken bauen und entscheiden, welches Datenbanksystem am Besten zu unserer Aufgabe passt und am leichtesten zu implementieren ist.

Wir haben uns für Strategie S2 entschieden und Prototypen entwickelt. Der Kunde wünscht, dass wir kein komplexes Datenbanksystem wie z.B. MySQL[2] verwenden, da solche eine komplexe Installation erfordern. Also bleibt die Möglichkeit einer "embedded" Lösung. Dafür kamen Derby[3], HyperSQL und SQLDroid in Frage. Nach dem Prototypenbau und dem Testen von beiden, haben wir uns dazu entschlossen HyperSQL auf der Serverseite und SQLDroid auf dem Client zu verwenden.

Verwandte Strategien

- S1:
  - 1.1. Strategie S1
  - 2. Strategie S2
- S2:
  - 2. Strategie S1
  - 7. Strategie S1
  - 8. Strategie S1

7	**In Java geschriebenen Webserver der SSL konfigurierbar ist**
Wir müssen einen Webserver finden, der in Java geschrieben ist und gut für die Nutzung von SSL zu konfigurieren ist.	
Involvierte Einflussfaktoren	

[2] http://www.mysql.com/
[3] http://db.apache.org/derby/

- Produktfaktoren:
  - 6. Verschlüsselung der Kommunikation
  - 8. Zugriff vieler Nutzer zur selben Zeit akzeptieren
  - 9. Import & Export des Datenbestandes
  - 10. Lauffähigkeit des Servers auf allen gängigen Betriebssystemen
  - 11. Fremdbibliotheken dürfen für den Einsatz in Forschung und Lehre keine Beschränkungen haben.
  - 13. Performanz der Implementierung
  - 14. Zuverlässigkeit der Implementierung
- Organisatorische Faktoren:
  - 17. Erfahrung des Teams
  - 18. Kenntnisse der Programmiersprache Java
- Technische Faktoren:
  - 24. Möglichst wenig Datenübertragung per Internet
  - 27. Nutzung eines SQL Datenbanksystems

Strategien

- S1: Wir erstellen von den verschiedenen Arten von offenen Webframeworks in Java, die eine SSL Kommunikation bieten, Prototypen und einigen uns auf den, der uns am meisten überzeugt.
- S2: Wir schreiben unseren eigenen Webserver, den wir dann für die Nutzung von SSL konfigurieren.

Wir haben uns für die Strategie S1 entschieden, denn einen eigenen Webserver zu implementieren ist ein zu hoher Aufwand und außerdem hat keines der Gruppenmitglieder die entsprechenden Kenntnisse dafür.
Entschieden haben wir uns für XMLRPC, da dies über HTTP läuft und wir uns selber kein eigenes Übertragungsprotokoll überlegen müssen. XMLRPC kann man als Java Servet in einem Java EE kompatiblen Server benutzen. Als offene Frameworks auf dem Server kamen für uns Jetty und Tomcat[4] in Frage. Ausgewählt haben wir schlussendlich dann Jetty.

Verwandte Strategien

[4] http://tomcat.apache.org/

- S1
  - 2. Strategie S1
  - 6. Strategie S2
  - 8. Strategie S1

8	**Verschlüsselte Kommunikation mittels SSL unter Android mit der Android API**

Die Kommunikation zwischen Server und der Applikation auf dem Endgerät muss verschlüsselt mittels SSL ablaufen. Da kein Gruppenmitglied vorher eben dies realisiert hat, müssen wir dies testen.

Involvierte Einflussfaktoren

- Produktfaktoren:
  - 4. Verschlüsselung der Kommunikation
  - 9. Import & Export des Datenbestandes
  - 11. Fremdbibliotheken dürfen für den Einsatz in Forschung und Lehre keine Beschränkungen haben
  - 13. Performanz der Implementierung
- Organisatorische Faktoren:
  - 18. Kenntnisse der Android Programmierung
- Technische Faktoren:
  - 22. Android als Betriebssystem

Strategien

- S1: Erstellen eines Prototypen, in dem wir die Kommunikation mittels SSL vom Server zum Endgerät testen und unsere Erfahrungen in den Architekturentwurf einfließen lassen.

Verwandte Strategien

- S1
  - 2. Strategie S1
  - 6. Strategie S2
  - 7. Strategie S1

9	**Zusammenführen der globalen Änderungen und der des einzelnen Nutzers**

Globale Änderungen auf dem Server und die lokalen Veränderungen der einzelnen Nutzer müssen sinnvoll und konfliktfrei zusammengeführt werden.

Involvierte Einflussfaktoren

- Produktfaktoren
  - 5. Implementierung von polyhierarchischen Bäumen
  - 6. Zusammenführen von Bäumen

Strategien

- S1: Komplettes überschreiben der lokalen Änderungen durch die Änderungen vom Server.
- S2: Der Benutzer kann entscheiden, ob er die lokalen Änderungen behält oder die des Servers komplett übernimmt.
- S3: Bei in-Konflikt-stehenden Änderungen die lokalen Änderungen der Nutzer durch die Änderungen auf dem Server überschreiben.
- S4: Bei Änderungen die in Beziehung zu einander kritisch sind, kann der Benutzer entscheiden welche übernommen werden sollen.

Wir entscheiden uns für die Strategie S2, denn es ist einfach zu implementieren und lässt dem Benutzer die Wahl.
Die anderen Strategien, sind entweder zu radikal (komplettes Überschreiben ohne Wahl) oder zu aufwändig zu implementieren, denn S3 und S4 bedürfen eines aufwändigen Algorithmus zur Erkennung von Zyklen.

Verwandte Strategien

- Keine

10	Authentifizierung der Administratoren
Wir müssen eine sinnvolle Weise finden eine Authentifizierung der Administratoren zu implementieren.	
Involvierte Einflussfaktoren • Produktfaktoren: – 7. Adminzugang wird benötigt	
Strategien • S1: Einen Admin-Login implementieren, mit Hilfe dessen sich die Administratoren auf dem Server einloggen können und anschließend Inhalte verändern können. • S2: Einen Admintoken fest ins Programm bzw. den Administrationszugang programmieren, mit dem der Admin authentifiziert wird. Wir haben uns für S2 entschieden, da diese Strategie leichter zu implementieren ist.	
Verwandte Strategien • Keine	

11	Änderungen an Vorschlägen
Wie geht unser Produkt damit um, wenn ein Nutzer seine Änderungen bzw. eigenen Karten online zur Verfügung stellt, aber ein anderer Nutzer an dieser Karte Änderungen vornimmt, bevor der Nutzer, der die Karte erstellt hat beim Nächsten Update erfahren hat, dass seine Karte angenommen wurde.	
Involvierte Einflussfaktoren • Produktfaktoren: – 5. Implementierung von polyhierarchischen Bäumen – 6. Zusammenführen von Bäumen	
Strategien	

- S1: Jeder Nutzer bekommt vom Server, beim Online-bereitstellen einer von ihm erstellen Karte eine ChangeTicketID zurück. Diese ChangeTicketID wird auf dem Server mit der entsprechenden Karte gespeichert. Ändert nun ein anderer Nutzer diese Karte, wird eine zweite ChangeTicketID generiert, an den Client übertragen und zusammen mit der alten ID gespeichert. Beim nächsten Update des erstellenden Nutzers überträgt dieser seine ChangeTicketID und auf dem Server werden alle entsprechenden Karten rausgesucht, die eine seiner ChangeTicketIDs enthalten. Zusätzlich zu den sowieso abonnierten Karten und Kategorien. So lässt sich sicherstellen, dass ein Client die von ihm erstellen Karten beim Update immer eindeutig zuordnen kann, selbst wenn der Name oder andere Attribute der Karte geändert wurden.

Verwandte Strategien

- Keine

12	**Darstellung der polyhierarchischen Bäume in der Datenbank**

Die zu implementierenden polyhierarchischen Bäume müssen in der Datenbank dargestellt werden.

Involvierte Einflussfaktoren

- Produktfaktoren:
  - 5. Implementierung von polyhierarchischen Bäumen
  - 6. Zusammenführen von Bäumen
- Technische Faktoren:
  - 27. Nutzung eines SQL-Datenbestandes

Strategien

- S1: Jede Kategorie hat eine Liste von Ober- und Unterkategorien. Damit besteht eine *n:m*-Beziehung. Alle Kategorien, die keine Oberkategorien besitzen, werden zur Wurzel.

Verwandte Strategien

- Keine

13	Zyklenfreiheit der Bäume
Es muss sichergestellt werden, dass es keine Möglichkeit für die Nutzer gibt, einen zyklischen Baum zu erstellen.	
Involvierte Einflussfaktoren • Produktfaktoren: – 5. Implementierung polyhierarchischen Bäumen – 6. Zusammenführen von Bäumen	
Strategien • S1: Es kann nur eine Änderung der Hierarchie von den Kategorien vorgenommen werden, wenn diese keinen Zyklus ergeben. Dies überprüfen wir vor jeder Änderungen mit einer Methode.	
Verwandte Strategien • Keine	

14	Import und Export von Daten
Daten müssen schnell und verlustfrei importiert und exportiert werden.	
Involvierte Einflussfaktoren • Produktfaktoren: – 3. Speichern und Lesen von Daten – 9. Import & Export des Datenbestandes • Technische Faktoren: – 26. Nutzung eines SQL Datenbanksystems – 27. Möglichkeit des Imports von CSV-Datein	
Strategien	

- S1: Vom Admintool aus kann der komplette Bestand der Daten als CSV-Datei exportiert werden. Ebenso kann kann man Daten mittels einer CSV-Datei importieren, Daten die auf dem Server sind werden dann überschrieben.

Verwandte Strategien

- Keine

15	**Möglichkeit zur Mehrsprachigkeit garantieren**

Wir müssen eine Lösung finden, wie wir leicht die Unterstützung mehrere Sprachen sicherstellen.

Involvierte Einflussfaktoren

- Produktfaktoren:
  - 0. Mehrsprachigkeit der Applikation

Strategien

- S1: Alle unsere Strings werden im von Android dafür vorgesehenen String.xml-Datein gespeichert. Für jede unterstütze Sprache gibt es eine eigene .xml-Datei, die anhand der Systemsprache ausgewählt werden kann. Ist für die Systemsprache keine eigene .xml-Datei vorhanden, wird auf die “normale” String.xml zurückgegriffen.

Verwandte Strategien

- Keine

16	**Eingeschränkte Auswahl der Fremdbibliotheken**

Da nur Fremdbibliotheken, die für Forschung und Lehre frei stehen, genutzt werden dürfen, sind wir in der Wahl der Fremdbibliotheken eingeschränkt.

Involvierte Einflussfaktoren

- Produktfaktoren:
  - 11. Fremdbibliotheken dürfen für den Einsatz in Forschung und Lehre keine Beschränkungen haben

Strategien

- S1: An Stellen, an denen wir auf Frameworks angewiesen sind, nehmen wir Open-Source Lösungen. So verwenden wir Jetty als Webserver. Als Datenbanksystem verwenden wir HyperSQL und SQLDroid.

Verwandte Strategien

- Keine

17	Einhalten der spezifizierten Ladezeiten

Die in der Anforderungsspezifikation angegebenen Höchstdauern der Ladezeiten, der Aktionen und ähnlicher Prozesse, müssen eingehalten werden.

Involvierte Einflussfaktoren

- Produktfaktoren:
  - 13. Performanz der Implementierung
- Technische Faktoren:
  - 23. Vielfalt an Zielhardware

Strategien

- S1: Teilweise liegt das Einhalten der Ausführungszeiten nicht in unserer Kraft, da es bei bestimmten Aktionen auf die gerade vorhandene Internetverbindung ankommt. Allerdings haben wir eine effiziente Möglichkeit gefunden Daten zu übertragen. Hierfür verwenden wir XMLRPC. Ansonsten versuchen wir unsere Applikation auf möglichst vielen Endgeräten zu testen, um sicherzustellen, dass unsere Activities performant genug sind um auch auf “schlechterer Hardware” schnell zu sein.

Verwandte Strategien

- Keine

18	Behandeln vieler Zugriffe zur gleichen Zeit

Der von uns mit ausgelieferte Server muss mit vielen gleichzeitigen Zugriffen durch viele Nutzer umgehen können und möglichst gut auf die ganzen Anfragen zu antworten.

Involvierte Einflussfaktoren

- Produktfaktoren:
  - 13. Performanz der Implementierung
  - 14. Zuverlässigkeit der Implementierung

Strategien

- S1: Wir versuchen den Umfang der Übertragenen Daten zu minimieren, dadurch können gleichzeitig mehr Zugriffe gewährleistet werden.
- S2: Wir versuchen die Zugriffe zu minimieren, sprich nur Zugriffe zum Server erlauben wenn es absolut nötig ist.
- S3: Wir setzen Serversysteme ein, die in der Lage sind auch unter hoher Last innerhalb gewisser Latenz zu antworten (Jetty in diesem Fall).

Wir haben versuchen die beiden Strategien in Einklang zu bringen, wobei eine Minimierung von übertragenen Daten und Zugriffen normalerweise im Gegensatz zueinander stehen.

Verwandte Strategien

- Keine

19	Einhalten der festgelegten Deadline

Wir haben eine festgelegte Deadline, die wir einhalten müssen, also muss unser Produkt vor dieser Deadline fertig sein. Je komplexer die Architektur, desto mehr Zeit werden wir für die Umsetzung dieser benötigen.

Involvierte Einflussfaktoren

- Im Prinzip wird dieses Problem von allen Einflussfaktoren bedingt, da potentiell jeder Einflussfaktor Einfluss auf die für die Fertigstellung des Produktes benötigte Zeit hat.

Strategien

- S1: Wir müssen eine umsetzbare Architektur erarbeiten, die eher einfach gehalten ist, da wir diese wegen Zeitmangel sonst vermutlich nicht umsetzten können.
- S2: Wir müssen effiziente Strategien zur Lösung der Probleme finden und diese umsetzen.

Wir verwenden klar die Strategie S1, da wir alles so einfach wie möglich halten wollen. Denn eine Aufwendige Architektur erhöht auch die Fehleranfälligkeit. Strategie S2 impliziert selber einen gewissen Zeitaufwand. Da versuchen wir ein gesundes Mittelmaß zu finden.

Verwandte Strategien

- Keine

20	**Probleme in der Gruppenarbeit**

Bei Gruppenarbeiten besteht immer die Gefahr von Streit oder interner Unruhe. Gerade in Verbindung mit Deadlines, also Arbeit unter Zeitdruck, kann es zu Streit kommen. Im schlimmsten Fall bedeutet dies, dass ein Gruppenmitglied oder Mehrere, die Gruppe verlassen. Die Veränderung der Gruppengröße hat Einfluss auf unsere Architektur, da diese Veränderung auch eine Veränderung der Mindestanforderung bedeutet. Daraus folgt möglicherweise eine vereinfachte Architektur.

Involvierte Einflussfaktoren

- Organisatorische Faktoren:
  - 15. Einhalten der Deadline
  - 19. Größe des Teams

Strategien

- S1: Um inneren Spannungen vorzubeugen machen wir regelmäßig Treffen außerhalb der Uni-Veranstaltungen.
- S2: Sollten Gruppenmitgliedern mit Leistungen oder dem Verhalten anderer Gruppenmitglieder unzufrieden sein, soll so etwas bei unseren wöchentlichen Treffen in der Uni diskutiert bzw. angesprochen werden.

Wir wenden beide genannten Strategien an, da es unbedingt sicherzustellen gilt, dass kein Mitglied die Gruppe verlässt.

Verwandte Strategien

- Keine

21	**Blockieren der GUI**

Es gibt mehrere Möglichkeiten die zum Blockieren einer GUI führen können. In unserem Fall handelt es sich dabei um den Verbindungsaufbau mit der Datenbank bzw. dem Server oder bei rechenintensiven Aufgaben.

Involvierte Einflussfaktoren
Keine direkten Einflussfaktoren.

Strategien

- S1: Eine mögliche Strategie ist es, Netzwerkverbindungen und rechenintensive Aufgaben in andere Threads, im Falle von Android handelt es sich dabei um AsynchTasks, auszulagern. Diese Kontrolle übernimmt der *TaskManager*. Der *TaskManager* ist eine Ansammlung aller der Operationen, die im Hintergrund durchgeführt werden können.
- S2: Alle Operationen die im Hintergrund laufen können, werden an der Stelle definiert, an der sie verwendet werden. Die Activities starten dann AsynchTasks.

Wir haben uns für Strategie S1 entschieden, da die Strategie S2 fehleranfälliger ist. Da nicht alle AsynchTasks an einem zentralen Ort verwaltet werden, kann es leicht passieren, dass Operationen doppelt ausgeführt bzw. aufgerufen werden. Außerdem würde Strategie S2 den Code der Activities unnötig "aufblähen".

Verwandte Strategien

- Keine

3 Konzeptionelle Sicht (Lukas)

3.1 Überblick

Unsere Architektur besteht aus drei grundlegenden Komponenten (siehe hierfür Abbildung **??**). Es gibt eine Serverkomponente, welche die nötigen Daten zu den Karteikarten und Kategorien speichert. Außerdem ist es über diese Serverkomponente möglich eine Kommunikation zwischen den Nutzern und den Administratoren des Systems zu etablieren.

Eine weitere Komponente ist das Admintool, mit dem die Administratoren den Datenbestand der Serverkomponente pflegen und Nachrichten empfangen und senden können. Dieses Admintool wird ein einfaches Konsolen-Tool und kommuniziert über eine Netzwerkverbindung mit der Serverkomponente. Dies bedeutet, dass das Admintool auf jedem Rechner, der eine Netzwerkverbindung[5] mit der Serverkomponente aufbauen kann, ausgeführt und benutzt werden kann.

Die dritte Komponente ist der Android-Client. Dieser stellt alle Funktionen bereit, die die Nutzer zum Lernen von Karteikarten benötigen, wie etwa das Erstellen von Karteikästen, das Lernen dieser Karteikästen, das Erstellen/Ändern/Löschen von Karten und Kategorien. Alle nötigen Daten für den Betrieb kann sich der Client von der Serverkomponente herunterladen. Außerdem implementieren wir in der Clientkomponente das Userinterface. Dieser Android Client kommuniziert ebenfalls über eine Netzwerkverbindung mit der Serverkomponente.

Ein dreigliedriges System zu entwickeln ist naheliegend, da wir genau zwei Gruppen von Nutzern haben und eine dritte Komponente benötigen, über die diese kommunizieren. Es wäre noch vorstellbar gewesen, das *Admintool* in die Serverkomponente zu integrieren. Dies vereinfacht die Architektur jedoch nicht deutlich und nimmt den Vorteil der Administration von überall, den unsere Architektur bietet.

3.2 Komponente: Serversystem

Die Komponente *Server* ihrerseits besteht aus vier weiteren Komponenten (siehe Abbildung **??**). Zum einen gibt es einen Java EE konformen Java Servet-Server[6]. Dieser nimmt HTTPS Verbindungen von den Android Clients und dem Admintool entgegen

[5] Genauer: XMLRPC über HTTPS

[6] Die Entscheidung hierfür resultiert aus der Strategie In Java geschriebener Webserver der SSL konfigurierbar ist:S1 und dem Problem Behandeln vieler Zugriffe zur gleichen Zeit

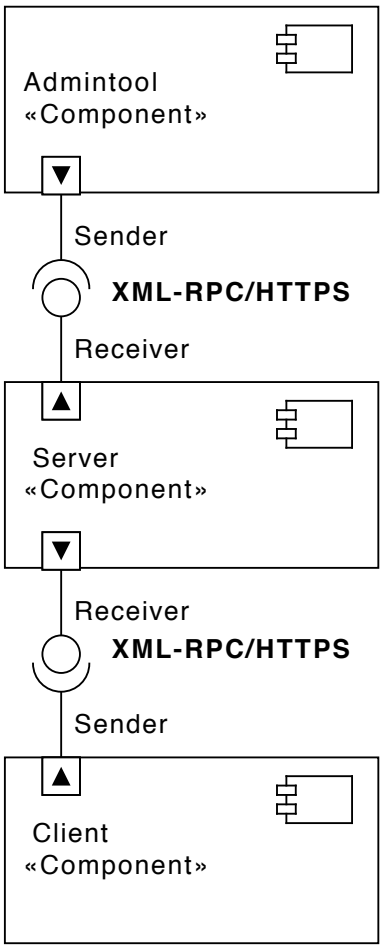

Abbildung 1: Übersicht der Komponenten

und gibt die Anfragen an eine weitere Komponente, das *XMLRPC-Servlet*, weiter. Dieses *XMLRPC-Servlet* stellt das Protokoll XMLRPC bereit, über das unsere drei Hauptkomponenten (siehe Abbildung **??**) kommunizieren.

Das *XMLRPC-Servlet* fragt über die Komponente *DB-Connector*, Daten aus der Datenbank ab und schreibt welche in diese. Diese *DB-Connector* Komponente stellt eine leichte Abstraktionsschicht über der Datenbank dar und implementiert wichtige Methoden zur Manipulation dieser. Die letzte Komponente ist die Datenbankkomponente *DB*. Diese ist die tatsächliche Implementierung des Datenbanktreibers, des von uns gewählten Datenbanksystems[7]

3.3 Komponente Admintool

Die Komponente *Admintool* (siehe Abbildung **??**) ist einfach gehalten und besteht aus nur zwei Komponenten. Das Konsolen-Interface parsed die Eingabe und setzt sie in entsprechende Aufrufe von Methoden um. Diese werden über die XMLRPC-Komponente über HTTP an die Serverkomponente übermittelt. Von dieser lädt die Komponente *Admintool*, Listen von Änderungen oder neue Nachrichten, übermittelt Daten, wenn diese aus einer CSV-Datei importiert werden, oder überträgt Datensätze, die in eine CSV-Datei exportiert werden. Außerdem können die Administratoren mit dem Admintool Nachrichten an alle Nutzer senden.

3.4 Komponente Android Client

Die *Client*-Komponente ist die komplexeste unserer Komponenten (siehe hierfür Abbildung **??**) und besteht aus weiteren fünf Komponenten. Die *GUI*-Komponente ist die Nutzerschnittstelle. Sie nimmt Eingaben von dem Nutzer entgegen und stellt die Karten und weitere Daten für den Lernfluss dar.

Wenn der Nutzer Eingaben tätigt die eine komplexe Operation erfordern und die möglicherweise eine gewisse Zeit zur Vollendung benötigen, wird diese Operation an die Komponente *Taskmanager* übergeben, welcher die Operation im Hintergrund ausführt. Sobald die Operation vollendet ist, teilt die Komponente *Taskmanager* der Komponente *GUI* dies mit. Die Entscheidung Operationen in den *Taskmanager* auszulagern resultiert aus der Strategie Blockieren der GUI:S1.

Zur Ausführung der Operationen greift die Komponente *Taskmanager* entweder über die Komponente *DB-Connector* (vgl. *DB-Connector* in Abschnitt **??** wobei sie nicht identisch sind) auf die Komponente *DB* zu, oder sendet über XMLRPC und die Komponente XMLRPC Aufrufe an die Serverkomponente.

Die Komponente *DB-Connector* ist insoweit nicht mit der aus Abschnitt **??** identisch, als dass sie leicht andere Methoden zur Manipulation der Datenbank bereitstellt. Genau die,

[7] Wir haben uns für die SQL-Bibliothek HyperSQL entschieden.

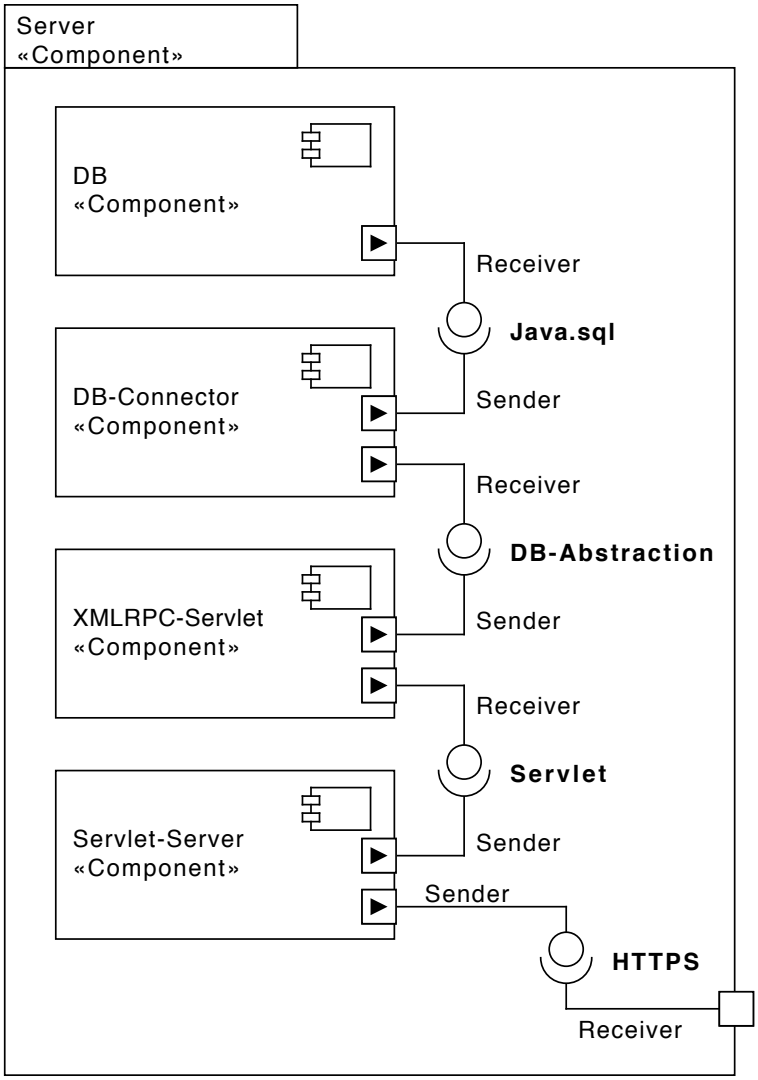

Abbildung 2: Diagramm der Komponente Server

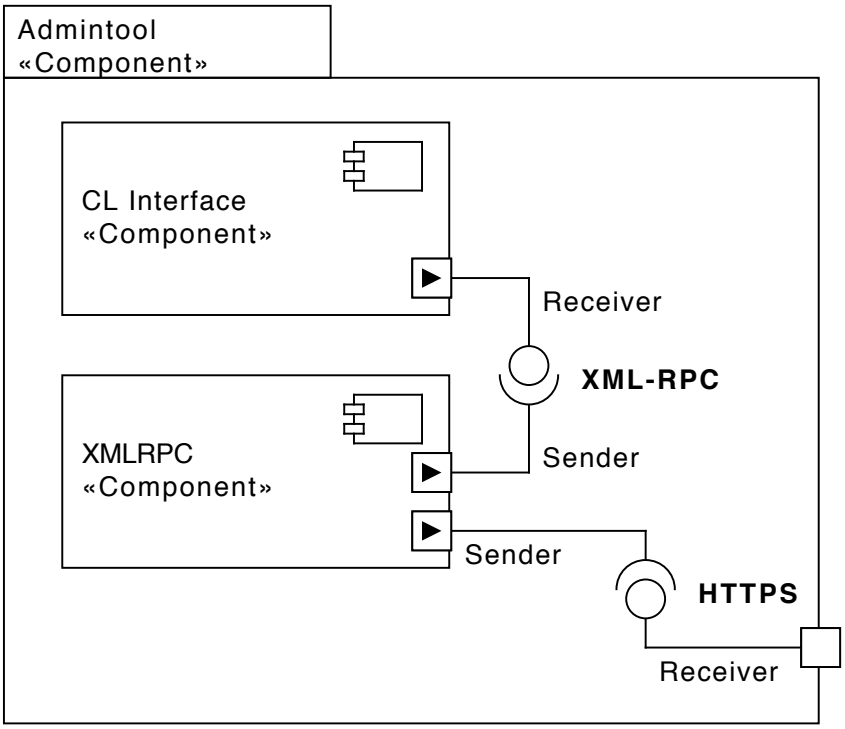

Abbildung 3: Diagramm der Komponente Admintool

welche die *Client*-Komponente für ihren Betrieb benötigt. Auch hier arbeitet die Komponente *DB-Connector* nicht direkt auf einer Datenbank, sondern über die Komponente *DB*, einen SQL-Datenbanktreiber.

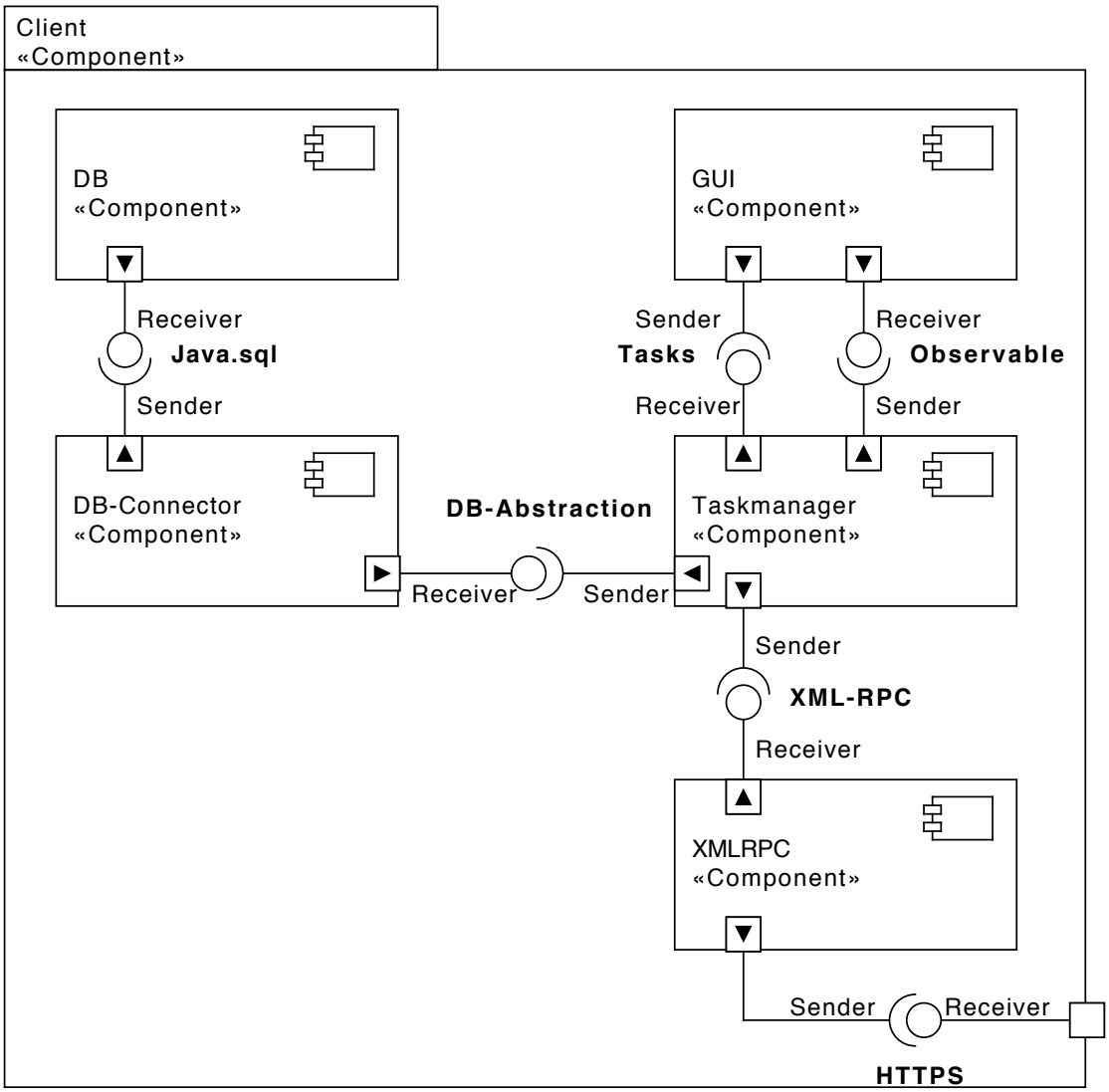

Abbildung 4: Diagramm der Komponente Client

4 Modulsicht (Lukas)

4.1 Protokolle

Protokolle im eigentlichen Sinn haben wir nicht entworfen. Die Kommunikation zwischen Client/Admintool und Serverkomponente erfolgt über XMLRPC. Dies ist ein Protokoll für *Remote-Procedure-Call*s, entfernte Methoden Aufrufe, eine Möglichkeiten die Ausführung von Methoden auf dem Server anzustoßen und den Rückgabewert zu erhalten. Dieses XMLRPC benutzt HTTP für die Übertragung der Daten, daher sollte die Kommunikation über jede Netzwerkschnittstelle möglich sein. Für die Verschlüsselung der Kommunikation wird SSL eingesetzt.

Die Schnittstellen zwischen Client/Admintool und Serverkomponente sind so entworfen, dass die Reihenfolge der Methodenaufrufe die korrekte Funktionalität und Ergebnisse nicht beeinflusst, insoweit, dass es nicht zu Fehlern kommen wird.

Die einzige Ausnahme ist die Klasse *Admintool* (siehe Abschnitt **??**). Hier benötigen die Methoden *applyChange()* und *declineChange()* eine *ID*, welche der Liste der Änderungen der Methode *listChanges()* zu entnehmen ist. Hierbei ist es jedoch lediglich sinnvoll die Methoden in der richtigen Reihenfolge aufzurufen, *applyChange()* und *declineChange()* funktionieren auch mit beliebigen *ID*s, solange diese gültig sind.

4.2 Pakete

Wir erstellen ein Paket *eu.ldots*. Dieses enthält weitere Unterpakete um gemeinsame Quellcodedateien zu bündeln. In Abbildung **??** sind diese vier Unterpakete aufgelistet. Sie orientieren sich an den in Abschnitt **??** genannten Komponenten. Die gemeinsamen Klassen werden dabei in ein Paket *common* gebündelt. Im Folgenden werden die Unterpakete näher beschrieben, die Erläuterungen zu den Klassen sind im Abschnitt **??** zu finden.

4.2.1 Common

Das Paket *eu.ldots.common* (siehe Abbildung **??**) bündelt alle Klassen die sowohl auf dem Android-Client, als auch auf der Serverkomponente Verwendung finden. In den Klassen dieses Pakets ist an weiterer Funktionalität nur welche aus dem Paket *java.util* notwendig.

Dieses Paket enthält folgende Klassen:

- CardBox.java
- AbstractContent.java
- CardObject.java
- CategoryObject.java

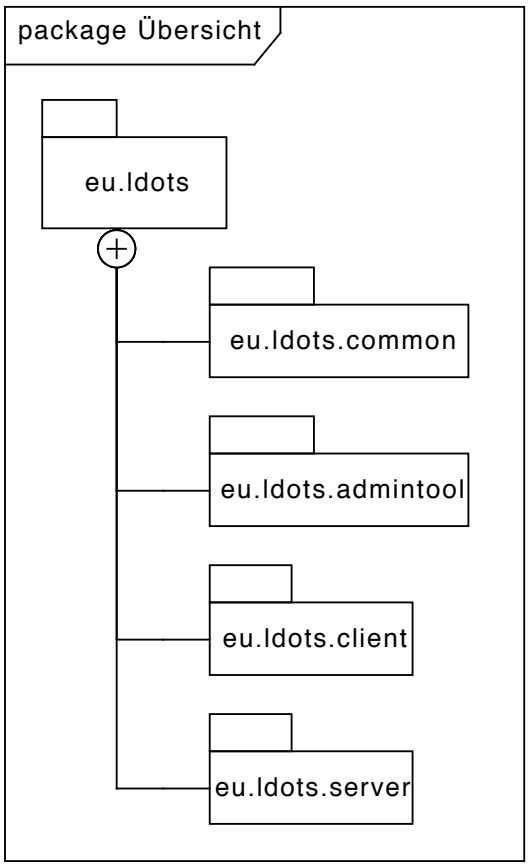

Abbildung 5: Paketdiagramm: Übersicht

- ChangeProposal.java
- DatabaseAbstraction.java

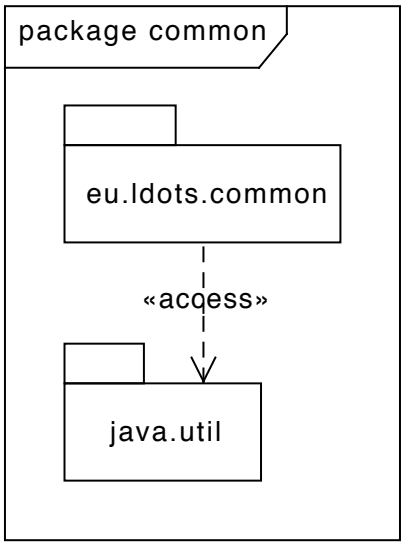

Abbildung 6: Paketdiagramm: Common

4.2.2 Cards

In dem Paket *eu.ldots.cards* (siehe Abbildung **??**) sind alle Klassen enthalten, um die Applikation auf dem Android-Gerät zu betreiben.

Für die Kommunikation mit der Serverkomponente wird die Bibliothek *org.xmlrpc.android* verwendet. Das Paket *android* enthält die Klassen des Android-SDK, *com.actionbarsherlock* ist eine Bibliothek, um die Actionbar auch in Android-Versionen <4.0 verwenden zu können. Das Paket *java.sql* ist für die Schnittstelle zur SQL-Datenbank notwendig, *java.util* für Listen und ähnliches.

Dieses Paket enthält folgende Klassen, welche die Android Aktivitäten darstellen. Diese entsprechen der Komponente *GUI* aus der Konzeptsicht (vergleiche Abbildung **??**).

- BaseActivity.java
- ActMain.java
- ActAddCardbox.java
- ActCategoryList.java
- ActComposeMail.java
- ActEditCard.java

- ActEditCategory.java
- ActGlossary.java
- ActLearnCard.java
- ActLearnSystemSettings.java
- ActUpdateSettings.java
- ActViewCard.java

Außerdem gibt es weitere Klassen, um die Logik, die auf dem Client benötigt wird, bereitzustellen. Die Zugehörigkeit zu den Komponenten in **??** ist offensichtlich.

- Taskmanager.java
- XMLRPCClientLayer.java
- DatabaseAbstractionClient.java
- LearningSystem.java
- LeitnerSystem.java
- TwoBoxSystem.java

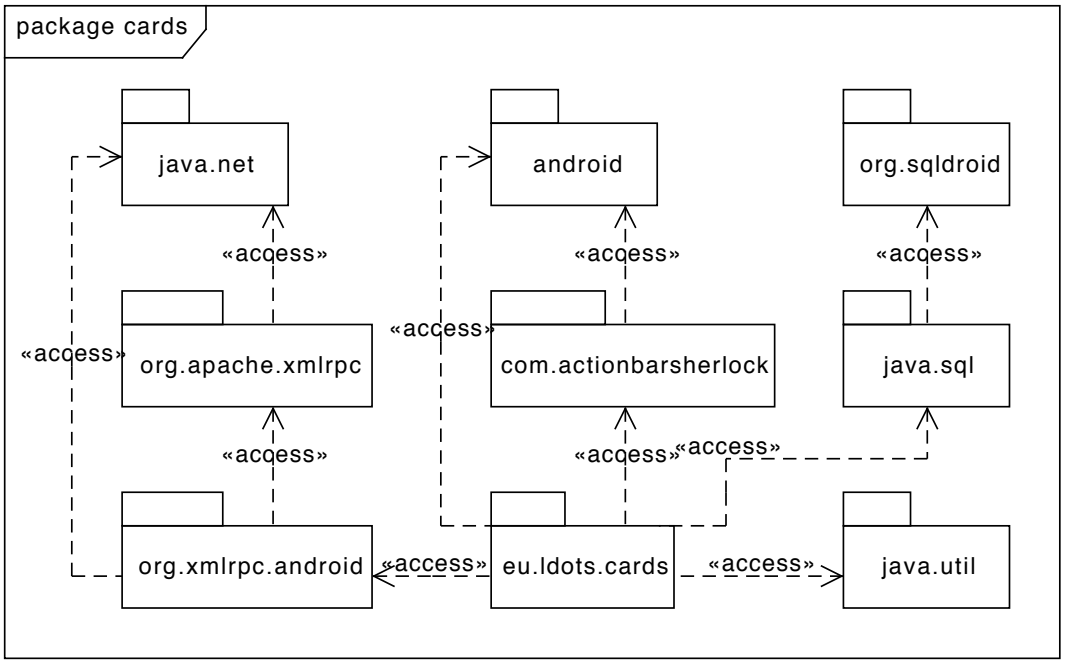

Abbildung 7: Paketdiagramm: Client

4.2.3 Server

Das Paket *eu.ldots.server* (siehe Abbildung **??**) enthält die Klassen *DatabaseAbstractionServer.java* und *XMLRPCServlet.java*. Diese beiden sind die Komponenten, die wir für die Serverkomponente selbst entwickeln müssen. Die anderen Komponenten, die für die Serverkomponenten, (siehe Abbildung **??**) bestehen aus dem Java Servet-Server Jetty und der Java Bibliothek *java.sql*. Für die Verwendung der Datenbank ist außerdem der Datenbanktreiber *org.hsqldb.jdbc* notwendig. Außerdem wird Funktionalität aus dem Paket *java.util* verwendet.

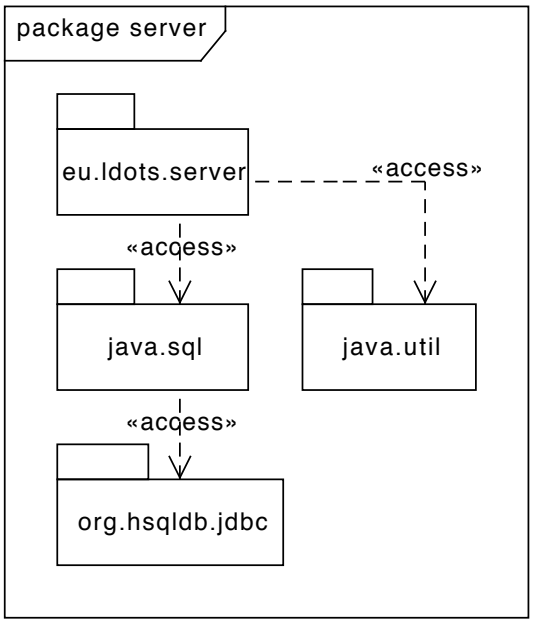

Abbildung 8: Paketdiagramm: Server

4.2.4 Admintool

Das Paket *eu.ldots.admintool* (siehe Abbildung **??**) enthält nur die eine Klasse *Admintool.java*. Diese benötigt für die Kommunikation mit der Serverkomponente die Bibliothek *org.apache.xmlrpc*, welche ihrerseits *java.net* verwendet. Auch hier wird wieder *java.util* verwendet.

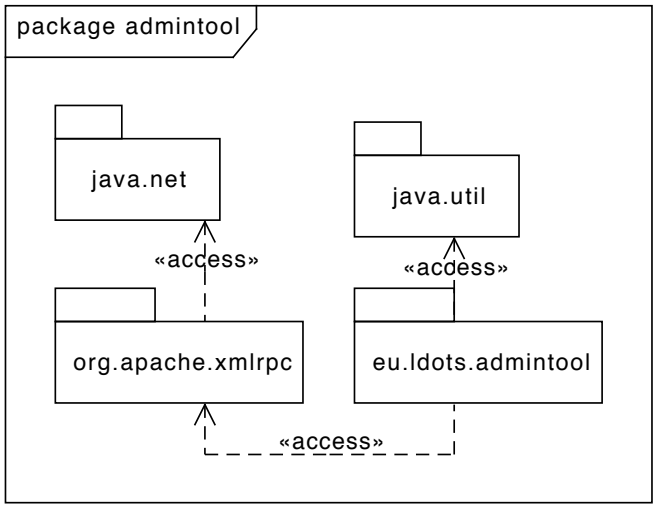

Abbildung 9: Paketdiagramm: Admintool

4.3 Klassen

4.3.1 Übersicht

Das vollständige Klassendiagramm ist in Abbildung **??** zu finden. Im Folgenden werden einzelne Teile genauer betrachtet. Dabei ist jeweils im Klassendiagramm markiert, auf welchen Bereich des Diagramms sich die Erläuterungen beziehen.

Abbildung 10: Klassendiagramm: Übersicht

4.3.2 Admintool

Das Admintool (siehe Abbildung **??**) ist ein recht einfach gehaltenes Modul. Ableiten von anderen Klassen oder Implementieren irgendwelcher Interfaces ist hier nicht erforderlich. Das Admintool stellt genau die Methoden bereit, die ein Redakteur zur Pflege des Systems benötigt. Er kann Nachrichten versenden und vom Server abholen, sich Änderungsvorschläge anzeigen lassen, bestätigen oder ablehnen und Datenbestände importieren und exportieren. Für dieses Modul entschieden haben wir uns aufgrund von Problem Import und Export von Daten und der Notwendigkeit, Änderungen und Nachrichten zu verwalten.

Im Admintool werden nicht umfangreich Daten manipuliert oder zwischengespeichert, daher sind nur drei Attribute zum Betrieb dieses Moduls notwendig: Die Adresse und der Port unter der der Server zu erreichen ist und ein geheimer Schlüssel, mit welchem sich das Admintool als solches beim Server kenntlich macht und so die Möglichkeit hat, geschützte Methoden auszuführen.

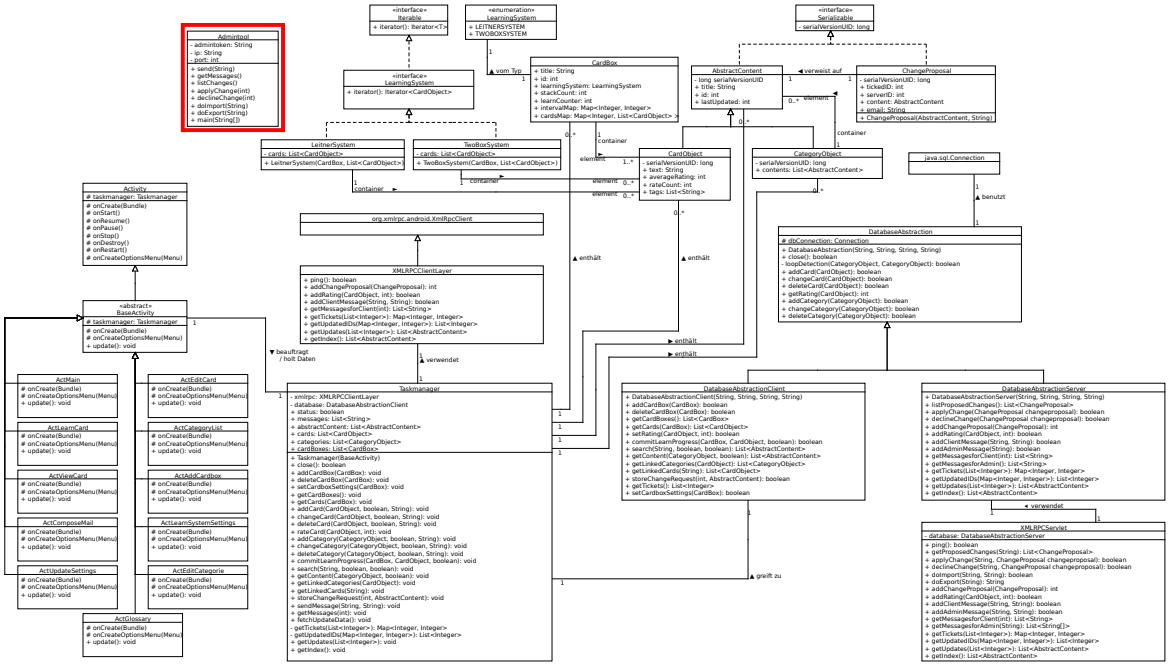

Abbildung 11: Klassendiagramm: Admintool

4.3.3 Datentypen

Unser System definiert diverse Objekte, die alle logisch zusammengehörende Daten bündeln (siehe Abbildung **??**). All diese Datentypen haben nur *public* Attribute, um diese einfach setzen und lesen zu können.

Alle Datentypen die per XMLRPC über eine Netzwerkschnittstelle gesendet werden,

implementieren um dies zu ermöglichen das Interface *Serializable*. *CardObject* und *CategoryObject* erben von *AbstractContent*, da wir einige Methoden definiert haben, die eine Liste zurückgeben, die Karten und Kategorien enthalten kann (beispielsweise *DatabaseAbstractionClient.getContent()*). Die Attribute ergeben sich teilweise aus den Problemen Änderungen an den Vorschlägen und Darstellung der polyhierarchischen Bäume in der Datenbank.

Eine *CardBox* speichert sich welches Lernsystem benutzt wird. Außerdem enthält sie eine Liste aller in ihr enthaltenen Karten, der Anzahl der Fächer und Informationen wie oft welches Fach gelernt werden soll. Aus den Karten aus einer *CardBox* und mit der Information um welches Lernsystem es sich handelt, kann eine Instanz einer der von *LearningSystem* abgeleiteten Klassen (*LeitnerSystem* und *TwoBoxSystem*) erstellt werden, welche die Karten richtig sortiert und einen Iterator über diese bereit stellt.

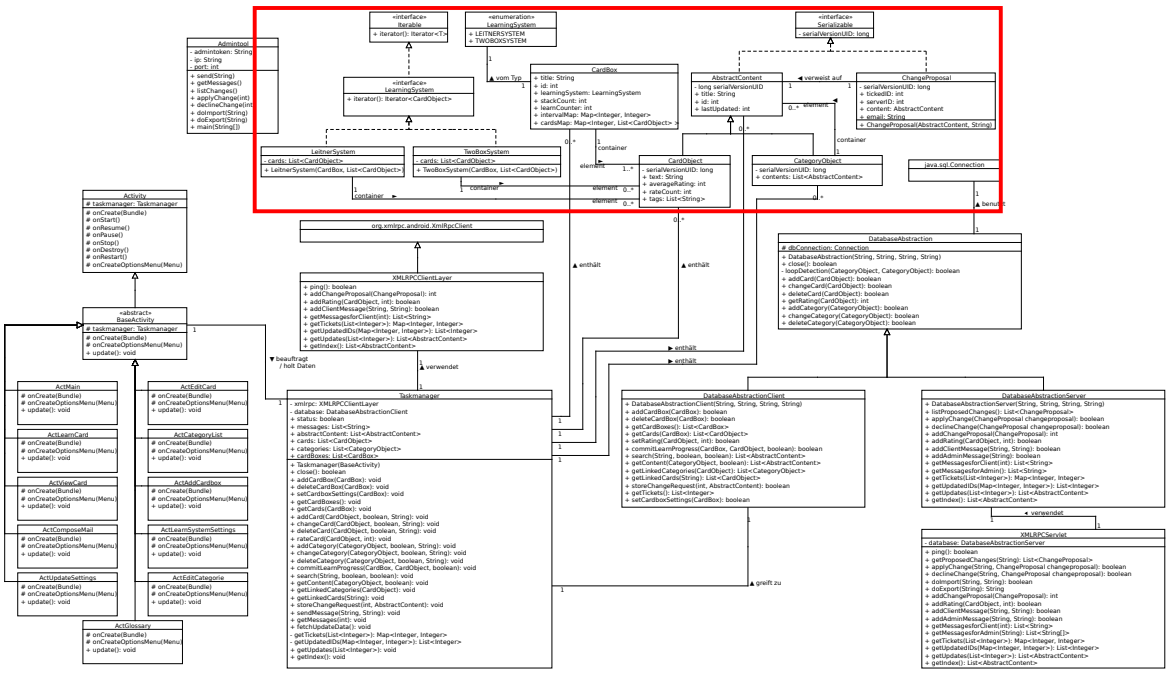

Abbildung 12: Klassendiagramm: Datentypen

4.3.4 Datenbankabstraktion

Diese drei Klassen (*DatabaseAbstraction*, *DatabaseAbstractionClient* und *DatabaseAbstractionServer*, siehe hierfür Abbildung **??**) abstrahieren von der rohen SQL-Datenbank und stellen Methoden bereit, um sie zu manipulieren. Wir haben diese Abstraktion eingeführt, damit sich nur einige Entwickler mit der Datenbank beschäftigen müssen (siehe Strategie Mangelnde Kenntnisse der Gruppenmitglieder über das Datenbankmanagementsystem:S2). Außerdem haben wir mit der *DatabaseAbstraction* ein Modul, in dem

beim Hinzufügen von Verbindungen zwischen Kategorien auf Schleifen geprüft werden kann (Problem Zyklenfreiheit der Bäume).

Die Verbindung zu der Datenbank erfolgt dabei über eine Instanz der Klasse *Connection* aus dem Paket *java.sql*. Gleiche Methoden werden in der Klasse *DatabasAbstraction* implementiert, *DatabaseAbstractionClient* und *DatabaseAbstractionServer* stellen Methoden bereit, die nur auf dem Android-Client beziehungsweise in der Serverkomponente benötigt werden.

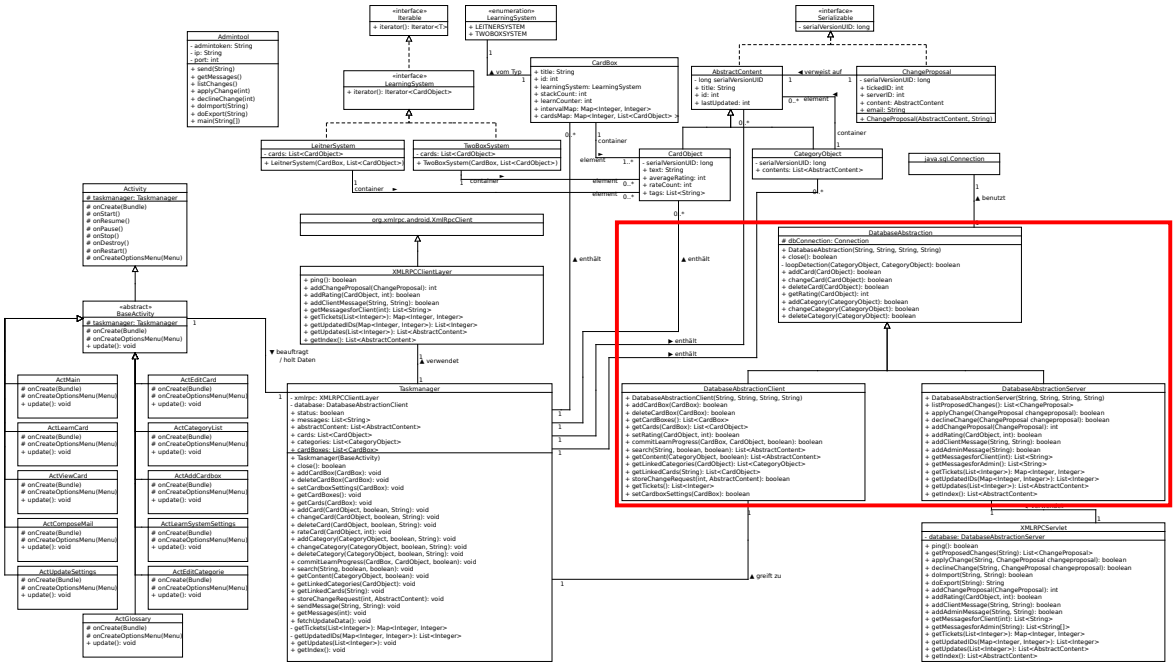

Abbildung 13: Klassendiagramm: Datenbankabstraktion

4.3.5 XMLRPC Server

Die Klasse *XMLRPC-Servlet* (siehe Abbildung **??**) stellt eine dünne Schicht vor der Klasse *DatabaseAbstractionServer* dar. Sie enthält alle Methoden der Datenbankabstraktion mit den gleichen Signaturen. Eine Ausnahme sind die geschützten Methoden (siehe folgende Auflistung). Diese lassen sich nur ausführen, wenn der richtige Schlüssel übergeben wurde.

- getProposedChanges()
- applyChange()
- declineChange()
- doImport()
- doExport()

- addAdminMessage()
- getMessagesforAdmin()

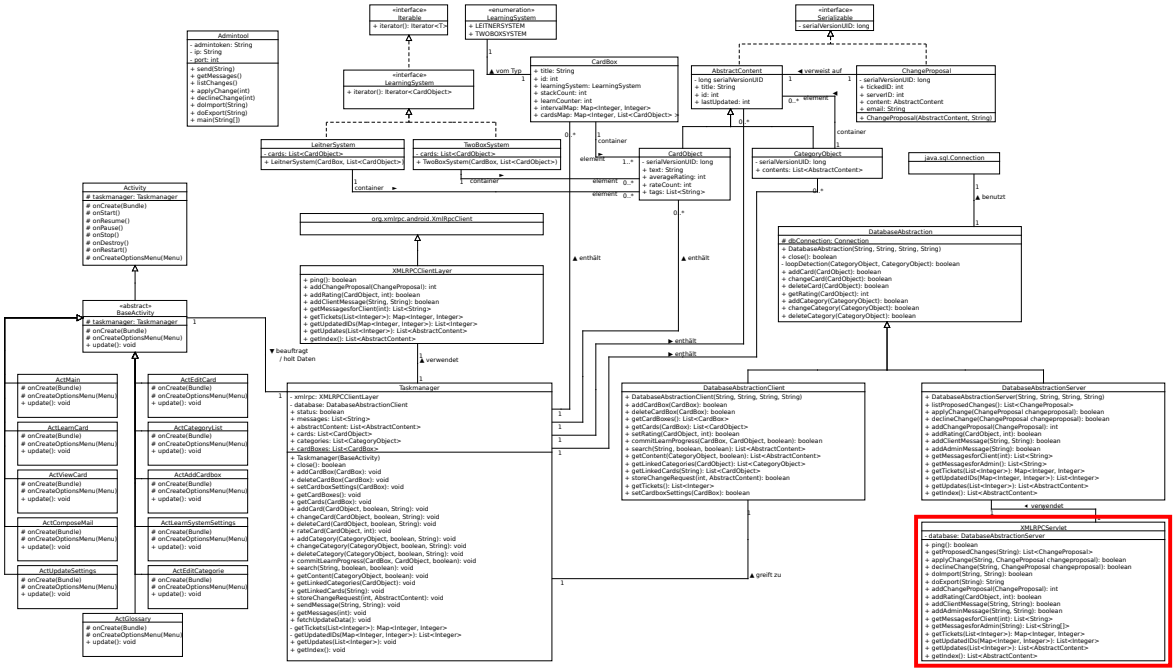

Abbildung 14: Klassendiagramm: XMLRPC Server

4.3.6 Aktivitäten

Der Teil in Abbildung ?? zeigt alle Aktivitäten. Jede Aktivitäten muss von *Activity*, der Activity-Basisklasse aus dem Android-SDK, erben. Jede Aktivität kann über die Klasse *Taskmanager* Operationen, die möglicherweise blockieren können und somit das *gui* zum Stocken bringen können, auslagern (siehe Problem Blockieren der GUI). Damit sie ein Ergebnis von der ausgelagerten Methode erhalten können, haben alle Aktivitäten eine Methode *update()*, durch welche ihnen mitgeteilt werden kann, dass die Operation die sie in Auftrag gegeben haben, abgeschlossen wurde.

Daher erben alle Aktivitäten von einer Basisklasse *BaseActivity* (welche auch ein Attribut zum speichern einer Referenz auf den *Taskmanager* enthält) und eben diese Methode *update()* definiert.

4.3.7 Taskmanager

Die Klasse *Taskmanager*, zu sehen in Abbildung ??, ermöglicht es, Operationen im Hintergrund zu starten und bei der Beendigung benachrichtigt zu werden. Die Methoden dieser Klasse werden verwendet um Methoden über XMLRPC aufzurufen, oder die lokale

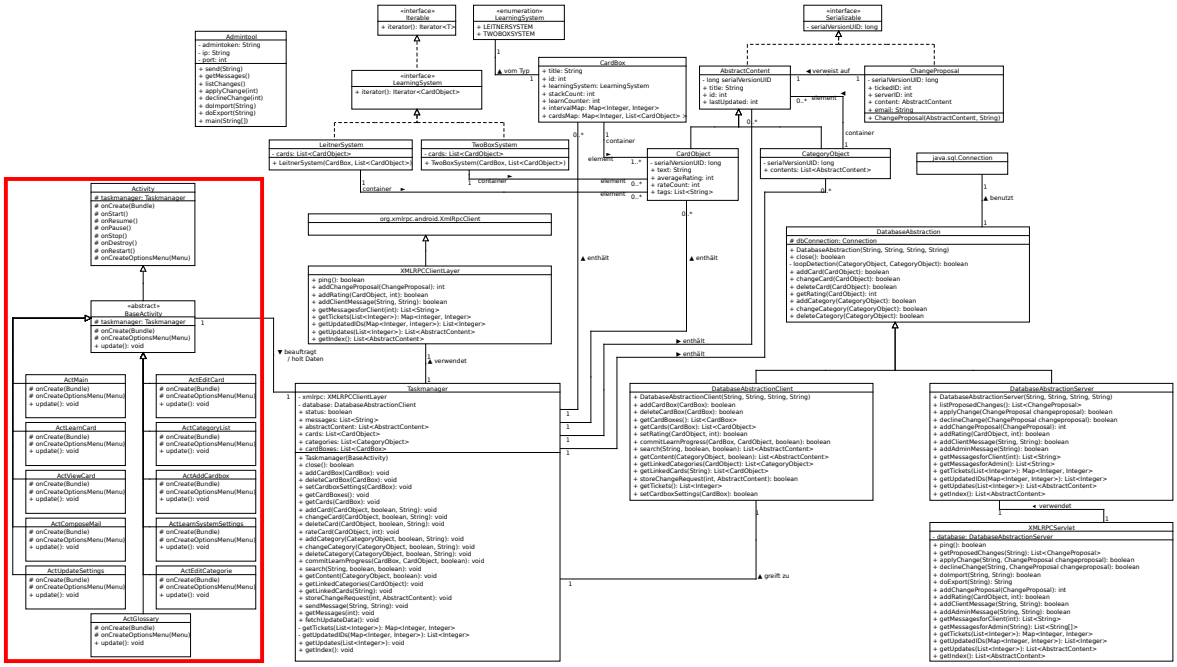

Abbildung 15: Klassendiagramm: Aktivitäten

Datenbank zu modifizieren. Dafür übergibt die Klasse, die Methoden des *Taskmanager* aufruft, bei der Instanziierung ihres Taskmanagers eine Referenz auf sich selbst, um bei der Beendigung benachrichtigt zu werden.

Die Kommunikation über XMLRPC geschieht über eine weitere Klasse: *XMLRPCClientLayer*. Dies ist bloß eine Klasse um die Handhabung von XMLRPC komfortabler zu gestalten (Parameter müssen der XMLRPC-Bibliothek als *object-Array* übergeben werden, Methoden geben nur *object* zurück).

Nachdem eine Aktivität eine Operation in Auftrag gegeben hat, wird die Abarbeitung dieser in den Hintergrund verlagert. Ist die Abarbeitung fertig, speichert der *Taskmanager* das Ergebnis in einer Instanzvariablen und ruft *update()* auf der Aktivität auf, welche die Operation gestartet hat. Diese kann sich nun das Ergebnis von der *public* Instanzvariablen holen.

5 Datensicht (Arne)

In der Datensicht von dem Server wird das Datenbanklayout dargestellt (siehe Abbildung **??**). Zentral geht es bei dem Server um die Speicherung der Karteikarten inklusive der dazugehörigen Kategorien. Daher nimmt dieser Teil auch den größten Teil der Datenbank ein. Die wohl wichtigste Tabelle ist die *cards*-Tabelle, da hier die Begriffe abgelegt werden. Es gibt es mehrere Attribute, welche den Karten zugeordnet sind. Hierzu zählen unter Anderem das Attribut ID, welches ein Autoincrementfeld darstellt und eine ein-

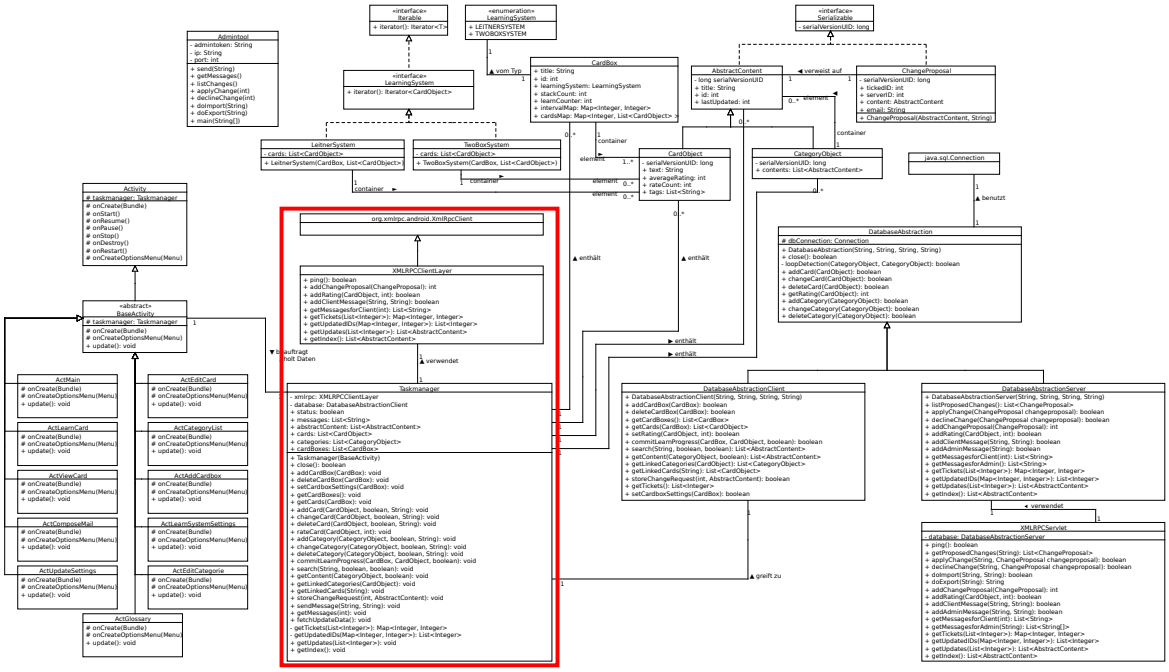

Abbildung 16: Klassendiagramm: Taskmanager

deutige ID für eine Karte ist.

Durch das automatische Hochzählen der ID durch die Datenbank, wird garantiert, dass die IDs einmalig sind und somit jeder Begriff bzw. jede Karteikarte anhand dieser ID identifiziert werden kann. Außerdem gibt es noch ein Attribut *title*, um die Bezeichnung der Karte / des Begriffs abzulegen. Dieses ist vom Typ VARCHAR (quasie ein String mit einer maximalen Länge von 255 Zeichen). Dazu kommt noch das Attribut *text*, welches den Inhalt der Karte abspeichert. Das Attribut *lastUpdated* ist ein Integer Feld mit einer Länge von 10 Zeichen, da hier ein Timestamp abgelegt wird, und zwar der von dem letzten Änderungszeitpunkt der Karte.

Dies ist wichtig für unseren Updateprozess der Clienten. Das Attribut *averageRating* speichert die aktuelle Bewertung der Karte durch den Benutzer. Da wir nur eine Bewertung im Bereich 0 bis 5 unterstützen, reicht hier ein Integer mit einem Zeichen. *ratingCount* ist ein Zähler, der bei jeder Bewertung der Karte um einen erhöht wird. Dieser ist für interne Zwecke und zum Nachvollziehen wie viele Benutzer die Karte bereits bewertet haben, interessant.

Die Zwischentabelle *cardsCategories* stellt die Verbindung zwischen Kategorien und Karteikarten her, denn hier werden die IDs der Karten, mit den IDs der Kategorien verbunden. Dadurch kann ein Begriff in mehreren Kategorien sein und mehrere Kategorien auch den selben Begriff enthalten.

Die Tabelle *categories* fällt etwas kleiner aus als jede für Karteikarten, denn hier wird nur der Titel für die Kategorie und ein Timestamp der letzten Aktualisierung benötigt.

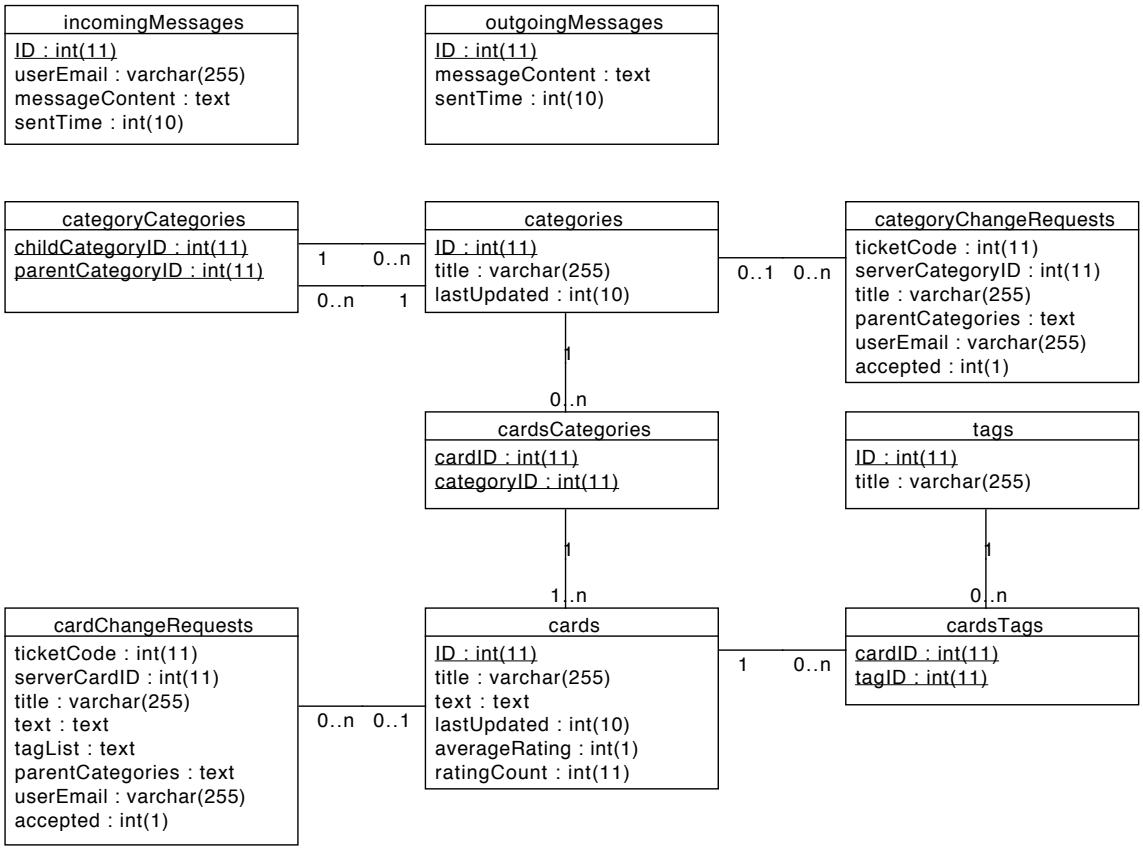

Abbildung 17: Datenbanklayout Serverseitig

Dies wird über die Felder *title* und *lastUpdated* gelöst. Des weiteren wird auch hier eine eindeutige Zuordnung benötigt. Daher gibt es auch hier das Feld ID mit einem Autoincrement, welcher die IDs der Kategorien unabhängig voneinander hochzählt. Hierdurch lässt sich jeder Kategorie eine eindeutige ID zuordnen und die Kategorie auch andersherum über die ID wieder identifizieren.

Damit mit der Datenbank eine Polyhierarchie gespeichert werden kann, gibt es nun noch eine „Zwischentabelle “für die Zuordnung von Kategorien zu Kategorien. Diese trägt den Namen *categoryCategories*, was so viel bedeutet wie: eine Kategorie ist mehreren anderen Kategorien zugeordnet. Die Zuordnung findet hier, auch wie bei Karteikarten zu Kategorien, über die IDs der Kategorien statt.

Damit den Karteikarten auch Tags zugeordnet werden können und trotzdem Redundanzen durch häufiger auftretende gleiche Tags vermieden werden, gibt es eine extra Tabelle für Tags. Die Tags werden in der Tabelle *tags* gespeichert. Da hier nur die Tags ohne weitere Informationen abgelegt sind, reicht ein Attribut mit dem Namen des Tags (*title*) und einer eindeutigen ID (auch über Autoincrement gelöst) für den Tag aus.

Über eine Zwischentabelle *cardsTags* können über die IDs der Karten und der Tags beliebig viele Tags beliebig vielen Karteikarten zugeordnet werden.

Da Benutzer eigene Änderungsvorschläge einreichen können, müssen diese auch in der Serverdatenbank hinterlegt werden, bis sie durch einen Administrator bearbeitet (angenommen oder abgelehnt) werden. Da Kategorien und Karteikarten, aufgrund der unterschiedlich benötigten Attribute, in unterschiedliche Tabellen aufgeteilt sind, werden auch die Änderungsvorschläge getrennt gespeichert. Für die Karteikarten gibt es die Tabelle *cardChangeRequests*. Hier gibt es ein Attribut *ticketCode*, welches als eine Art Primärschlüssel agiert. Diese darf es jeweils nur einmal geben (zusammen mit der ticketID von den Kategorieänderungsvorschlägen). Die Ticket-ID wird generiert, indem alle ticketIDs von den cardChangeRequests **und** categoryChangeRequests, betrachtet werden und die nächst freie Nummer größer 0, als Ticket ID genommen wird.

Dies stellt sicher, dass eine TicketID nur einmalig ist. Das Attribut serverCardID speichert die ID der Karteikarte in der Serverdatenbank, falls der Datensatz schon im Server vorhanden ist, oder wenn die Änderung angenommen wurde und somit eine ID in der Serverdatenbank zugeordnet wurde. Das Feld *title* enthält den geänderten Namen bzw. Titel der Karteikarte. Das Feld *text* den geänderten Beschreibungstext der Karteikarte. In dem Attribut *tagList* wird eine durch Komma separierte Liste von Tags für die Karteikarte hinterlegt. Selbiges ist bei dem Attribut *parentCategories* der Fall, jedoch werden hier Kategorienamen hinterlegt, in denen die Karte nach der Änderung liegen soll.

Dies hat den Vorteil bei den Tags und Kategorien, dass wenn sie auf dem Server noch nicht vorhanden sind, diese ggf. erstellt und der Karteikarte zugeordnet werden können, ohne die anderen Tabellen vor Änderungsannahme zu berühren.

Das Attribut *userEmail* hinterlegt eine E-Mail-Adresse von dem Benutzer, der die Änderung eingereicht hat. Diese kann beim Abarbeiten der Änderungsvorschläge von einem

Administrator eingesehen werden. Zusätzlich haben wir noch ein Attribut *accepted* mit einem ein-Zeichen-langen Integerwert, welcher auf 1 gesetzt wird, wenn der Änderungsvorschlag durch einen Administrator angenommen wurde, damit sich der Client, der die Änderung eingereicht hat, diese auch herunterladen kann. Wird die Änderung abgelehnt, wird die ganze Zeile der Änderung in der Datenbank einfach gelöscht.

Die Tabelle *categoryChangeRequests* hat die gleiche Funktion, jedoch für Kategorien, daher sind hier die Felder leicht unterschiedlich, um den benötigten Attributen gerecht zu werden.

Des weiteren soll es möglich sein, als Benutzer Nachrichten an die Serveradministratoren zu schicken und umgekehrt, als Administrator Benachrichtigungen an alle Benutzer zu schicken.

Hierzu werden noch zwei weitere Tabellen benötigt. In der Tabelle *incomingMessages* werden die Nachrichten von Benutzern an die Administratoren eingespeichert. Das Attribut *ID* mit Autoincrement ist hier auch wieder dazu da, um eine eindeutige Zuordnung herzustellen. In dem Feld *userEmail* wird die E-Mail-Adresse des Senders hinterlegt. Das Attribut *messageContext* speichert den Nachrichteninhalt selbst und zu guter Letzt, wird in *sendTime* ein Timestamp vom Sendezeitpunkt hinterlegt. Nach Abruf der Nachrichten durch einen Administrator werden diese aus der Datenbank entfernt, um die Datenbank sauber zu halten.

Die Tabelle *outgoingMessages* (Benachrichtigungen von Administratoren an alle Benutzer) funktioniert auf die gleiche Weise, jedoch fehlt das Attribut *userEmail*, da Administratoren keine E-Mail-Adresse hinterlegen müssen. Der Timestamp dient bei den Benachrichtigungen dafür, dass ein Client nur Nachrichten seit der letzten Aktualisierung der Benachrichtigungen abholt.

Die Datenbank des Clients (siehe Abbildung **??**) ist im Großen und Ganzen ein Abbild der Serverdatenbank, daher ist auch die Beschreibung der gleichnamigen Tabellen die gleiche, wie bei der Serverdatenbank. Es wurden jedoch einige Attribute weggelassen, die für den Benutzer uninteressant sind. Die *changeRequest*-Tabellen enthalten nur noch die Zuordnung von der lokalen Datensatz-ID und der Ticket-ID, welche durch den Server generiert wurde, damit der Client sich die Änderung noch nach Annahme oder Ablehnung durch einen Administrator zuordnen kann und diese mit den vorhandenen lokalen Daten in Verbindung bringen kann. Des weiteren wurden die *Messages*-Tabellen weggelassen, da diese für den Clienten unerheblich sind, denn diese Daten sollen nicht persistent gespeichert werden.

Erweitert wurde die Clientdatenbank um die Karteikästen, die sich jeder Benutzer lokal anlegen kann.

Hierzu gibt es eine Tabelle *cardboxes*, in welcher die Namen der Karteikästen (*title*) und der dazugehörigen Lernsystemeinstellungen hinterlegt werden. Da wir vorerst nur zwei Lernsysteme anbieten, reicht ein Integer mit einem Zeichen für das Attribut *learningSystem*. Es kann für das Leitnersystem die Anzahl der Lernstapel angepasst werden, daher gibt es noch ein Attribut *stackCount*, welches diese hinterlegt. Auch kann für jeden Lern-

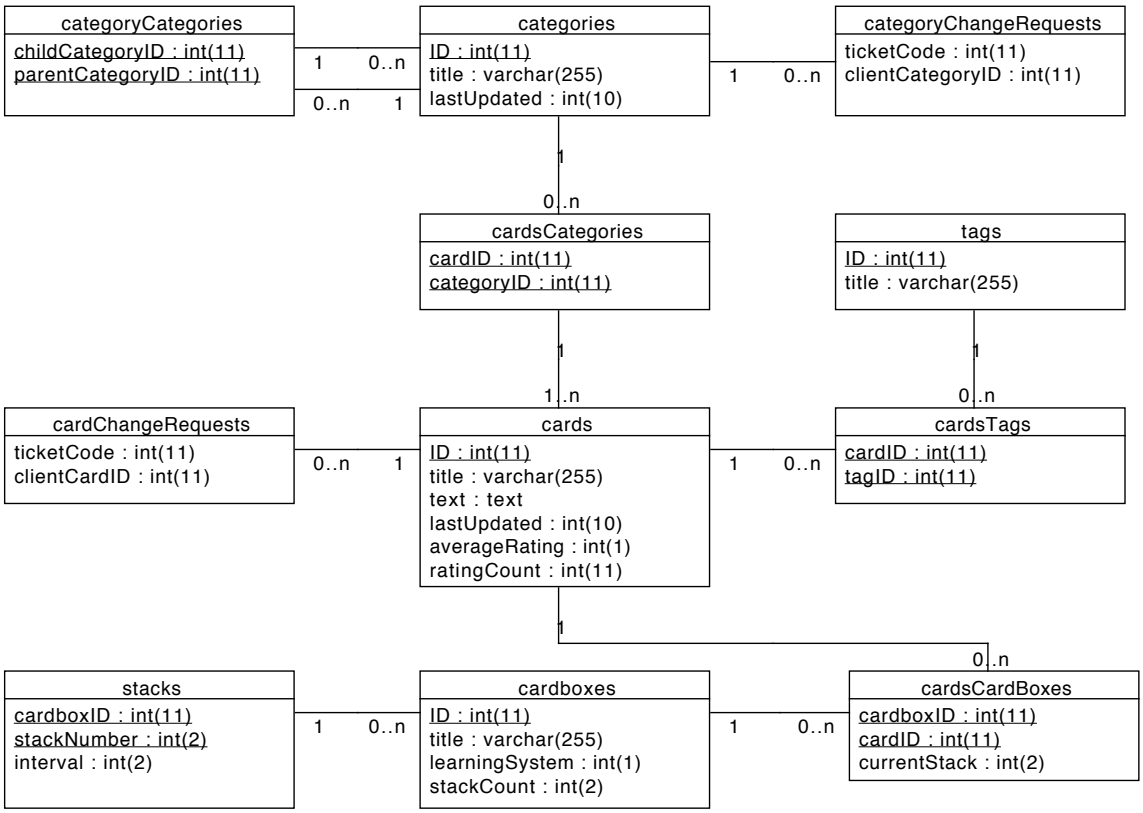

Abbildung 18: Datenbanklayout Clientseitig

stapel festgelegt werden, in welchem Intervall diese im Lernprozess wiederholt werden. Aus diesem Grund gibt es noch eine extra Tabelle *stacks*, wo das *Intervall* (Wiederholungsintervall), einer bestimmten Stapelnummer, in einem Bestimmten Karteikasten zugeordnet wird. Die Zuordnung findet hier über die ID des Karteikastens (einmalig, da Autoincrement) und der Stapelnummer statt.

Um jetzt noch einzelne Karteikarten einem Karteikasten zuzuordnen, gibt es auch hier eine Zwischentabelle. In dieser können beliebig viele Karten beliebig vielen Karteikästen zugeordnet werden. Zusätzlich wird noch in dem Attribut *currentStack* die Lernstapelnummer hinterlegt, auf welchem sich die jeweilige Karte gerade in dem bestimmten Karteikasten befindet. Dies ist wichtig für den Lernprozess.

6 Ausführungssicht (Thomas)

Abbildung **??** zeigt die Ausführungssicht der Architektur. Auf dem Server läuft ein Prozess *HyperSQL*. Dies ist das Datenbank Backend. Des weiteren existiert ein *Jetty* Prozess welcher unsere Servlet ausliefert. Bei jeder Verbindung vom Client auf den Server über XMLRPC wird eine Instanz des Moduls *XMLRPCHandler* und *DatabaseAbstarctionServer* erzeugt.

Der Server ist über *TCP/IP* mit dem Client (der Applikation auf dem Smartphone oder Tablet) und dem Admintool verbunden. Es können potenziell mehrere Clients und/oder Admintools mit dem Server verbunden sein.

Auf dem Smartphone/Tablet existiert eine Prozess der Applikation (*App*). In diesem Prozess gibt es das Modul *Activity*, welche in diesem Diagramm die GUI darstellt und jede mögliche Activity dieser Architektur sein kann. Des weiteren existieren die Module *DatabaseAbstractionClient*, *Taskmanager* und *SQLite*. Das *Taskmangermodul* kann als *innere Klasse* einen *AsyncTaskOperation* enthalten. Das Modul *SQLite* spiegelt das Datenbank Backend der Android Applikation wieder.

Das Admintool läuft auf einer *Workstation*. Der Prozess *Admintool* beinhaltet nur ein Modul *Admintool*.

7 Zusammenhänge zwischen Anwendungsfällen und Architektur (Thomas)

7.1 Karte ändern (119)

Im Sequenzdiagramm **??** ist folgender Sachverhalt dargestellt.

Der Benutzer bestätigt durch das Drücken des Buttons *Speichern* im Popup (siehe Anforderungsspezifikation, Anwendungsfall 119) das lokale Speichern und Senden der Karte an den Server. Durch den Aufruf von *onClick()* des Button wird ein *Taskmanager* Objekt

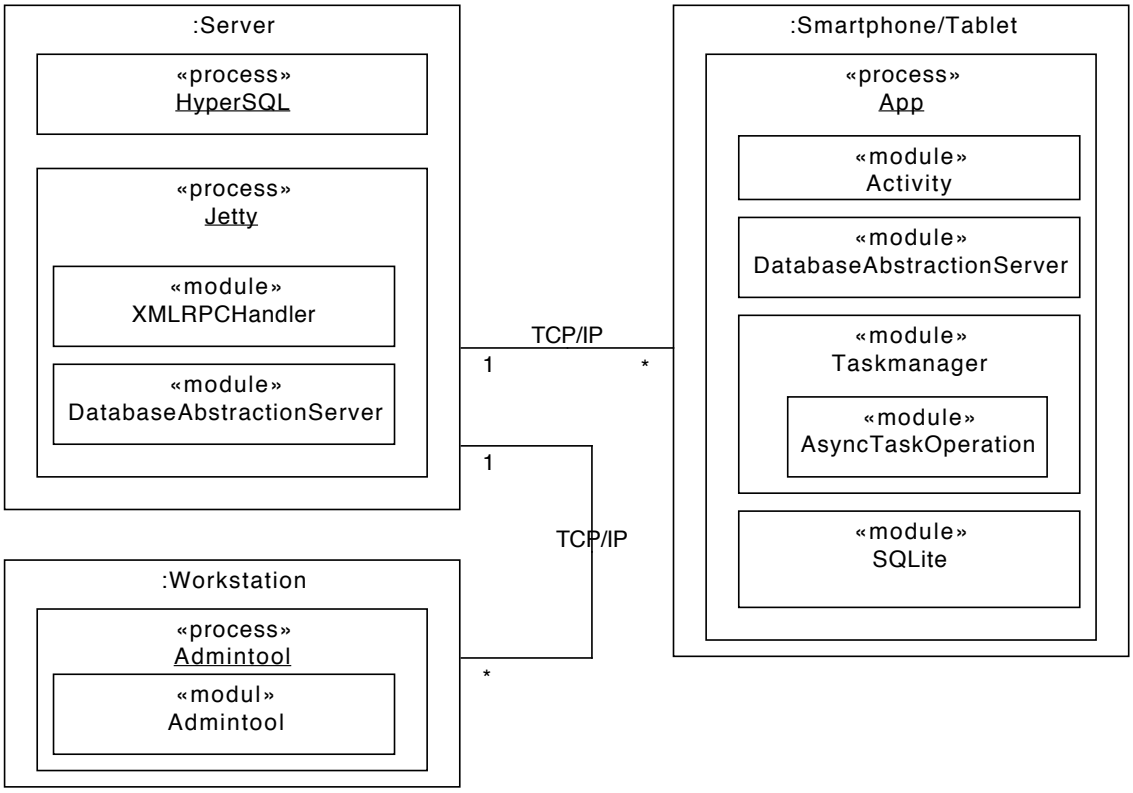

Abbildung 19: Ausführungssicht

erzeugt und darauf die Methode *changeCard(card, email)* aufgerufen. Es wird ein Objekt *changeProposal* vom Typ *ChangeProposal* aus den Daten *card* und *email* erzeugt.

Nun erzeugt der Taskmanager seinerseits ein *AsyncTask* Objekt. Dieses ruft er mit der Methode *execute()* auf. Der *AsyncTask* ruft in seiner Methode *doItInBackground()* die Methode *addChangeProposal(changeProposal)* des *XMLRPCServlet* auf. Dieser Aufruf passiert über einen *Remote Procedur Call* (*RPC*) auf der Serverseite. Hier ruft der *XMLRPCServlet* mittels der Methode *addChangeProposal(changeProposal)* das Objekt *DatabaseAbstractionServer* auf, welches er zuvor erstellt hat. Diese Methode speichert die Änderungen der Karte in der Datenbank und liefert die zugewiesene *ticketID* zurück. Der *XMLRPCServlet* übergibt diese *ticketID* an den *AsyncTask*. Diese ruft *storeChangeRequest(ticketID, card)* auf dem *DatabaseAbstarctionClient* auf. Diese Methode liefert bei erfolgreichem Eintragen in die Datenbank *true* zurück. Der *AsyncTask* wird beendet.

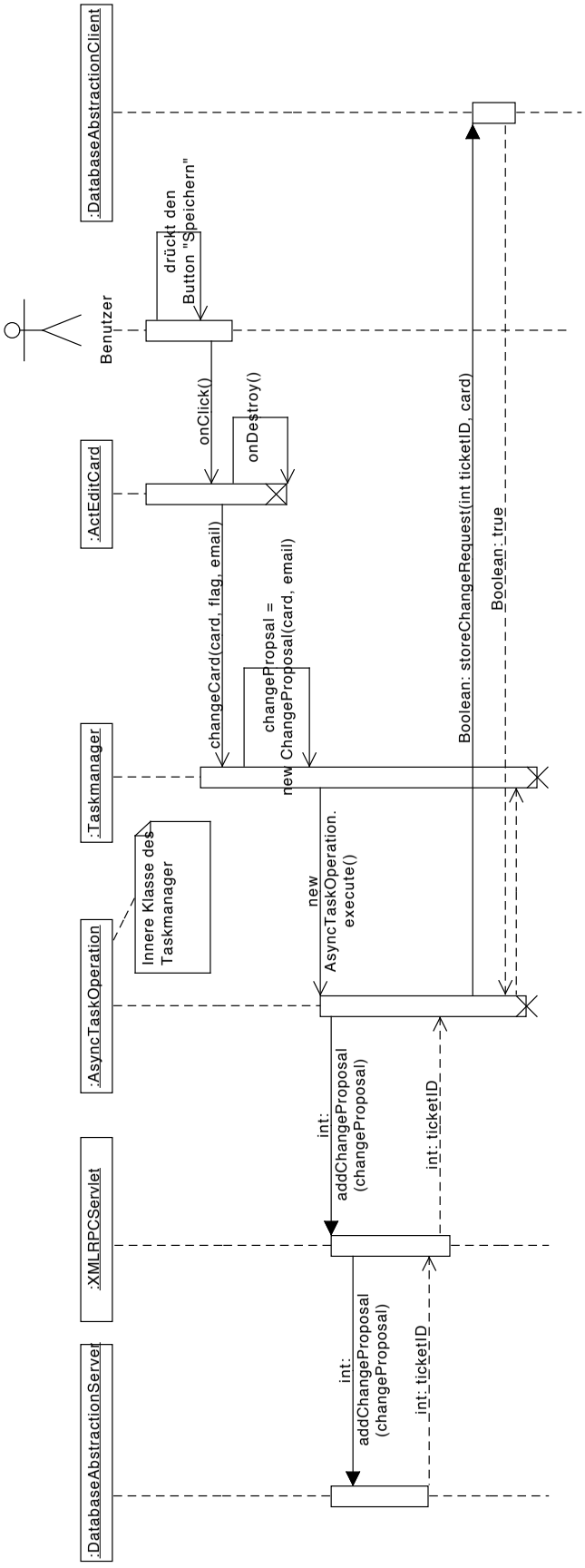

Abbildung 20: Sequenzdiagramm zum Anwendungsfall 119 Karte Ändern

7.2 Alle Updates laden (106b)

Im Sequenzdiagramm **??** ist folgender Sachverhalt dargestellt.

Mit dem Öffnen der Aktivität *ActUpdateSettings* wird ein *Taskmanager* Objekt erstellt und die Methode *fetchUpdateData()* des Taskmanager aufgerufen. Der *AsyncTask* mit dem Namen *AsyncTaskOperation*, welcher als innere Klasse des *Taskmanager* existiert, wird mittels *execute()* aufgerufen. Die *AsynctaskOperation* ruft in ihrer Methode *doItInBackground()* den *XMLRPCServlet* mit der Methode *getTickets(changeTickets)* auf und gibt dieser mit *changeTickets* die auf dem Client aktuellen "Change Requests", welche die aktuellen Änderungen am lokalen Inhalt darstellen, mit. Das *XMLRPCServlet*, auf der Serverseite, ruft die Methode *getTickets(changeTickets)* der *DatabaseAbstractionServer* auf. Die Rückgabe, ein *Map<Integer, Integer>*, ist eine Zuordnung der Change-Ticket-ID des Clients, auf die Karten oder Kategorie ID des Servers. Diese Rückgabe erhält der *AsyncTask*. Mit dieser Information und der Information aller lokalen Karten (und deren IDs auf dem Server), wird ein Objekt *contentServerIDs* erzeugt und die Methode *getUpdatedIDs(contentServerIDs)* des *XMLRPCServlet* aufgerufen. Das *XMLRPCServlet* ruft nun seinerseits auf der Serverseite die Methode *getUpdatedIDs(contentServerIds)* der *DatabaseAbstarctionServer* auf. Die Rückgabe beider Aufrufe ist eine Liste von Server IDs der geänderten Inhalte. Durch die Methode *update()*, aufgerufen auf die Aktivität *ActUpdateSettings*, wird nun das ListView der Aktivität aktualisiert, d.h., geänderte Inhalte werden im ListView markiert.

Der Benutzer bestätigt nun durch das Drücken auf die Ikone "Alle Updates laden" (siehe Anforderungsspezifikation, Anwendungsfall 106b) das Laden aller geänderten Inhalte. Es wird die Methode *getUpdates(List<Integer>contentServerIds)* des für diesen Zweck erzeugten Taskmanager aufgerufen. Dieser erzeugt mit *execute()* wieder einen *AsyncTask*. Die *AsynctaskOperation* ruft nun mittels XMLRPC die Methode *getUpdates(contentServerIds)* des *XMLRPCServlet* auf. Das *XMLRPCServlet* ruft seinerseits die gleiche Methode der *DatabaseAbstractionServer* auf. Die Rückgabe ist eine *List<AbstractContent>*, welche eine Liste über geänderte Inhalte darstellt, also alle geänderten Inhalte beinhaltet. Die *AsynctaskOperation* läuft nun in einer Schleife über diese Liste und fügt deren Inhalte in die Client Datenbank ein. Dazu entscheidet sie je nach Eintrag der Liste, ob es sich um ein *CardObject* oder ein *CatagoryObject* handelt. Je nachdem ruft der *AsyncTask* also die Methoden *addCard(CardObject)* oder *addCategory(CategoryObject)* der *DatabaseAbstractionClient* auf. Der erfolgreiche Eintrag eines Elementes in die Datenbank wird mit *Boolean: true* bestätigt. Nach jedem Eintrag wird das ListView der Aktivität *ActUpdateSettings* mittels *update()* aktualisiert.

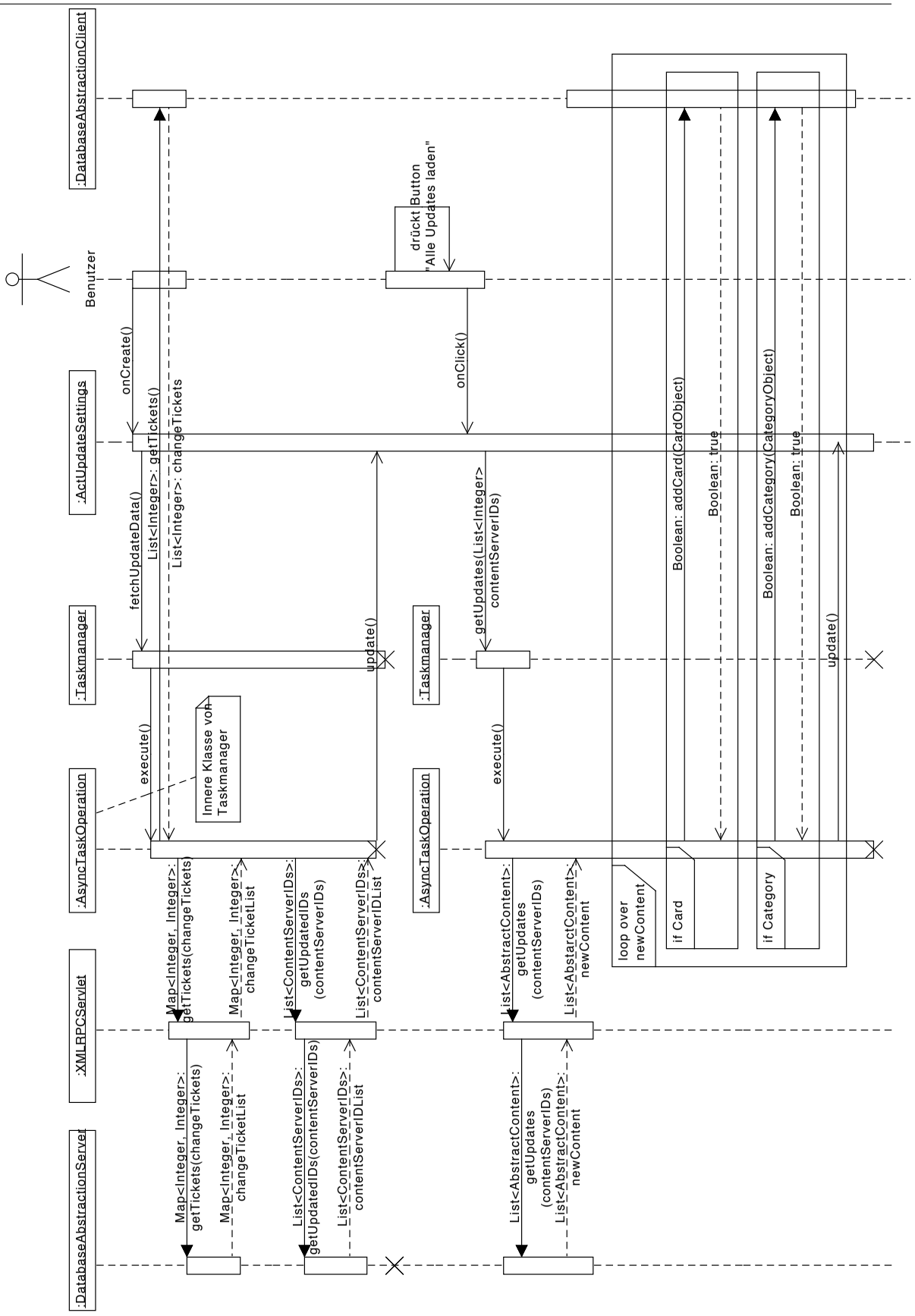

Abbildung 21: Sequenzdiagramm zum Anwendungsfall 106 Update

8 Evolution (Dominic)

Zusammenführen von globalen und lokalen Änderungen Beim Update, wo die globalen Änderungen vom Server übernommen werden und der Benutzer lokale Änderungen gemacht hat, hat er die Wahl zu entscheiden, was er behalten möchte. Die Änderungen, die der Server vorschlägt oder die lokalen Änderungen, die der Benutzer getätigt hat.

Bei genügend Zeit, oder wenn ein Algorithmus dafür vorgegeben wäre, der die globalen Änderungen und die des Benutzers sinnvoll zusammenführt, würden wir dies gerne umsetzen. Denn dies würde die Qualität des Produktes erheblich erhöhen.

Admin-Interface Wir haben beim Admintool auf eine GUI verzichtet, da dies nur eine optionale Anforderung ist und wir der Zeit wegen darauf verzichten.

Falls wir mehr Zeit haben und noch mit optionalen Features unsere Applikation aufwerten wollen, ziehen wir in Betracht eine GUI zu entwerfen.

Datenbank Momentan benutzen wir HyperSQL, auf Grund der Mindestanforderung, dass wir nur leichtgewichtige Datenbanksystem verwenden dürfen. Wenn sich später diese Anforderung lockert und wir ein Datenbanksystem unserer Wahl benutzen dürfen, wäre es denkbar hier MySQL[8] zu benutzen, da diese mehr Features bietet.

GUI Optimierung für Tablets Aus Gründen der Zeit haben wir die Optimierung der GUI für Tablets vernachlässigt. Unsere Applikation wird auf Tablets laufen, dafür wird diese eventuell nicht sehr anschaulich.

Wenn wir mehr Zeit hätten, könnten wir diese investieren, um auch für Tablets eine ansprechende GUI zu entwerfen.

Optimierung am Verbindungsverhalten Derzeitig gehen die Daten während des Sendevorgangs bei einer Verbindungsunterbrechung verloren und die Datenübertragung muss bei Wiederaufbau der Internetverbindung von vorne begonnen werden. Hier könnte man, wenn wir mehr Zeit hätten, einen Buffer einbauen, der das Gesendete zwischenspeichert. Dieser würde dazu dienen, bei Wiederaufbau der Internetverbindung, an der Stelle, wo diese unterbrochen wurde weiterzumachen.

Mehrsprachigkeit Wir haben in unserer Architektur, wie in den Mindestanforderungen gefordert, die Möglichkeit auf Mehrsprachigkeit in unserer Applikation berücksichtigt. Wenn wir mehr Zeit haben, könnten wir auch direkt eine weitere Sprache neben Deutsch implementieren.

[8] http://www.mysql.com/